Hindawi Publishing Corporation
Advances in Multimedia
Volume 2009, Article ID 905890, 20 pages
doi:10.1155/2009/905890

*Research Article*

# In-Network Adaptation of Video Streams Using Network Processors

**Mohammad Shorfuzzaman, Rasit Eskicioglu, and Peter Graham**

*Department of Computer Science, University of Manitoba, Winnipeg, MB, Canada R3T 2N2*

Correspondence should be addressed to Mohammad Shorfuzzaman, szaman@cs.umanitoba.ca

Received 23 October 2008; Revised 26 January 2009; Accepted 15 April 2009

Recommended by Yong Pei

The increasing variety of networks and end systems, especially wireless devices, pose new challenges in communication support for, particularly, multicast-based collaborative applications. In traditional multicasting, the sender transmits video at the same rate and resolution to all receivers independent of their network characteristics, end system equipment, and users' preferences about video quality and significance. Such an approach results in resources being wasted and may also result in some receivers having their quality expectations unsatisfied. This problem can be addressed, near the network edge, by applying dynamic, in-network adaptation (e.g., transcoding) of video streams to meet available connection bandwidth, machine characteristics, and client preferences. In this paper, we extrapolate from earlier work of Shorfuzzaman et al. 2006 in which we implemented and assessed an MPEG-1 transcoding system on the Intel IXP1200 network processor to consider the feasibility of in-network transcoding for other video formats and network processor architectures. The use of "on-the-fly" video adaptation near the edge of the network offers the promise of simpler support for a wide range of end devices with different display, and so forth, characteristics that can be used in different types of environments.



## 1. Introduction

The rapid growth of distributed computing and the Internet has led to demand for collaboration over wide area networks. This demand has been only partially met by existing multimedia and collaborative applications such as video-on-demand, teleconferencing, and telemedicine, which use the Internet for communication. For many such applications, group communication is a core component and the timely transfer of various types of media streams is a requirement.

Multicasting [1] is one of the building blocks of many collaborative applications and provides efficient communication between a single sender and multiple receivers. Messages originating from the sender are duplicated in the network as they are routed to the receivers that constitute the multicast group. Messages are forwarded through the use of a tree of routers called a "multicast tree" that is rooted from the sender, or possibly another predetermined point in the network, and which contains all multicast destinations (i.e., receivers) as leaves.

Initial efforts at implementing multimedia and collaborative applications for well-connected, high-end devices have proven to be successful (ivs [2], nv [3], vat [4], and vic [5] are widely used video and audio conferencing tools deployed over the Internet and the multicast backbone (MBONE) [6]). However, the usability of these new applications has been limited by a number of problems. One problem is how to deal with heterogeneity in the Internet. This heterogeneity, resulting from an increasing variety of networks and end systems, poses new challenges in communication support for collaborative applications. For example, consider a scenario where receivers have end systems ranging from simple, low-power Personal Digital Assistants (PDAs) to high performance workstations. Due to limited processing capabilities or slow network links, low-end receivers may not be capable of handling the same video streams as high-end receivers. Thus, different users in a group may have different requirements with respect to Quality of Service (QoS).

Multicasting performs one-to-many transmission so video is normally transmitted at the same rate to all receivers

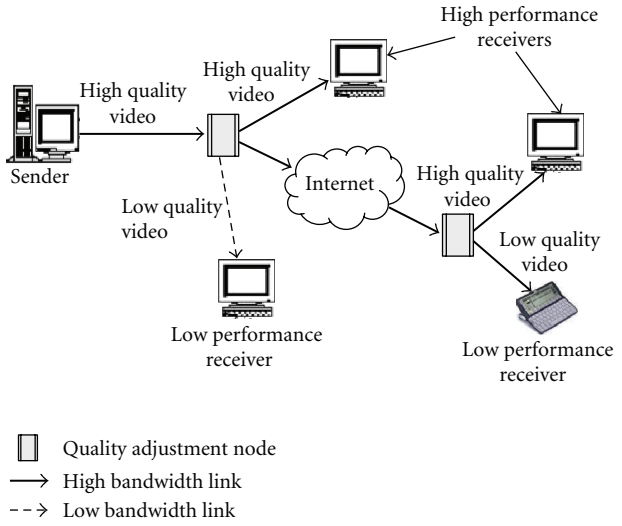


Figure 1: In-network transcoding in heterogeneous multicasting.

independent of their network attachment and end systems equipment. This means the source can only generate data at a rate that meets the capability of the most constrained receiver, although receivers having high bandwidth links would be capable of receiving correspondingly higher quality video streams. Additionally, not all the video streams possess equal value to all recipients since receivers may have different levels of interest in the incoming video streams. Unfortunately, most existing collaborative applications are not capable of capturing and exploiting user interest and thus must transmit the same, full quality, video streams to all participants. This approach results in resource wastage. Further, ignoring receivers' interests may also result in some receivers' quality expectations being unsatisfied since bandwidth may be wasted on unimportant streams.

Video adaptation (or transcoding) is a viable solution to these problems. Video streams originating from the source can be transcoded (i.e., modified) dynamically in the network according to the requirements of heterogeneous receivers and the capacity of their access links "downstream" in the multicast tree. Figure 1 illustrates in-network transcoding. Three high performance and one low performance receivers are connected to the video-quality adjustment nodes (i.e., routers) through high and low bandwidth links. The adjustment nodes dynamically adapt the rate of an incoming stream to meet the requirements of the receivers and network links they deliver to.

Work on active networks [7] has identified a number of issues that suggest that an active network-based approach to dynamic adaptation of video streams could be beneficial. Active networks allow users to inject customized programs into the network nodes and also support individual packets being programmed to perform specific actions as they traverse through the network. In the active networks paradigm, these packets are called "capsules" and they carry not only data but also references to the routines to be invoked at the nodes through which a capsule will pass. In this network model, programmability migrates from the application layer to the network layer and the network and application layers are, essentially, bridged together. Active network services running at the network layer can also exploit such information as knowledge of network topology and load conditions to achieve greater efficiency while application-level schemes can only use indirect metrics like data loss rate to speculate on network conditions.

In this paper, we seek to support the use of a wide range of end devices, varying connection characteristics and different user interests through the use of in-network video transcoding techniques implemented near the network edge in a fashion similar, but not limited, to that of active networks. The edge of the network being the boundary between the core network infrastructure and access network equipment provides an ideal location for doing video adaptation. In our system, video adaptation is done using a video adaptation node. The architecture of our prototype video adaptation node conforms to the node architecture provided by the active networks "reference model" [8].

The recent development of network processors has been motivated by the desire to support high data processing speed and greater programmability (for flexibility) in the network. Such devices offer an ideal environment for deploying the proposed system and this research contributes by helping to determine whether or not they are sufficiently powerful to support in-network video transcoding. Our current implementation uses the IXP1200 network processor for implementing the nodes that transcode MPEG-1 (MPEG-1 was chosen due to the limited instruction store and processing capability of the IXP1200.) video data to a desired bit rate. The functionalities provided by the active video adaptation node are arranged according to an active network architecture [8]. We implement our video adaptation process as an Active Application (AA), one of the major components of an active node. Capsules are not used.

We use requantization and selective frame dropping as transcoding techniques to adapt the video streams. Requantization is the process of dequantizing the Discrete Cosine Transform (DCT) coefficients of the video stream and then "requantizing" them with a new quantization step size to reduce the bit rate. In frame-dropping, frames that are not referenced by any other frames in the video sequence are dropped to keep the generated bit rate of a video stream from exceeding the allocated channel bandwidth and to reduce the frame rate. We evaluate our transcoding techniques in terms of transcoding latency, throughput, and accuracy. Finally, we provide some simple extrapolation of our results for MPEG-1 using the IXP1200 to different encoding schemes and more powerful network processors.

The remainder of this paper is organized as follows. Section 2 provides related work, Section 3 overviews videocoding techniques, and Section 4 reviews video adaptation algorithms. Section 5 and 6 discuss a simple low-pass filter and frame resizing, respectively. Section 7 presents a brief discussion of the use of network processors for video transcoding, which is followed by Section 8, which reviews and compares current network processor architectures. Sections 9 and 10 present the implementation details and

experimental results for our system, respectively. Finally, the paper concludes in Section 11 and discusses some directions for future work.

## 2. Related Work

A number of approaches to dynamically adaptive video multicast have been proposed in recent years aiming to address various issues and challenges including network heterogeneity in the Internet. The approaches can generally be divided into two categories. The first category adopts layered video [9–12] at the source (where video is composed of multiple layers with a base layer providing the lowest quality video signal, and each additional layer enhancing the quality of the base layer). Layered video is transmitted over multiple multicast sessions where each session corresponds to one layer. A receiver may subscribe to as many sessions as can be effectively supported by its processing capacity and link bandwidth. The second category uses video filtering/transformation [13–16] inside the network to produce a video stream of the desired quality "on the fly."

These existing adaptation approaches have a number of drawbacks and are subject to network heterogeneity problems. First, the video stream used in layered multicast schemes has to be layer encoded which makes such schemes restrictive and incompatible with most existing video applications. Second, the use of receiver-driven resource reservation in some approaches [15] leads to suboptimal use of network resources due to the lack of receivers' knowledge about the current network load and the varying nature of the network load over time. For example, if the receiver makes its reservation during a busy period, the network can only provide limited resources leaving the receiver with poor video quality even if the load is mitigated later. Third, the packet discarding technique used in some approaches [10] to handle network congestion is regarded as flawed [17] as it does not provide the best overall performance in terms of bandwidth utilization and end-to-end QoS. Instead of dropping a packet or sending it over a congested link by dropping high-frequency coefficients of video streams [14], the packet could be forwarded through a suboptimal route to attain better overall performance and efficient use of bandwidth. Fourth, many approaches focus only on single specific aspects of the problem. For example, Akamine et al. [13] describe a technique for the construction of multicast trees to be used in video transmission that satisfy different QoS requirements, but their work focuses only on which nodes should do video filtering. How the filtering mechanisms are implemented is not discussed. Finally, most work focuses only on congestion and bandwidth issues while the varying preferences of clients and the heterogenous characteristics of clients' devices also make group communication with multicast difficult.

Yamada et al. [16] present an active network-based video-quality adjustment method for heterogenous video multicast using the IXP1200 network processor. This was the first effort to use network processors in adaptive video multicasting. They only implemented a low-pass filter as a quality adjustment technique for real-time multicasting of MPEG-2 video. Their system, as described, cannot perform the required video adjustment at an acceptable rate.

Addressing these problems in collaborative applications requires an efficient mechanism for in-network adaptation of video streams, which influences user perceived quality and resource requirements both for the end-systems and the network. As described, though several application level schemes have already employed dynamic adaptation, none of them provides a complete solution to the problems faced by collaborative applications. Responding to this issue, the problem addressed in this paper is "how can network performance in collaborative applications be improved by detecting and managing preferences from the receivers for use in dynamic, in-network adaptation of data streams?" To this end, we discuss a framework that not only addresses network heterogeneity by considering clients' network connections and device characteristics but also supports delivering activity-based user interest hints into the network and using those hints at routers to adapt to changing user requirements through the dynamic modification of data streams.

## 3. Different Video Coding Standards

This section reviews video coding standards with a focus on the major video compression techniques. The characteristics of these standards play a key role in building practical video adaptation systems. Understanding the differences between MPEG-1 and the other standards provides a basis for extrapolating from our MPEG-1 results to other formats.

*3.1. MPEG-1 Video Coding.* In MPEG-1, video is represented as a sequence of individual still images consisting of a two-dimensional array of picture elements (pels). MPEG-1 video compression employs two basic coding techniques: intraframe coding and interframe coding.

In intraframe coding, spatial redundancy in the same video frame is reduced by DCT-based frequency transformation. In interframe coding, similarity between pels in adjacent frames, temporal redundancy, is reduced by motion compensation (MC). MPEG-1 divides the frames in a sequence into three types: intracoded frames (I-frames), forward predicted frames (P-frames), and bidirectional predicted frames (B-frames). An I-frame is encoded by intraframe coding without any reference to past or future frames. P and B frames are encoded using interframe coding. P-frames are coded with respect to the temporally closest preceding I-frame or P-frame. B-frames are encoded with respect to immediately adjacent I or P-frames past, future, or both.

An MPEG-1 video bitstream is divided into six layers: the video sequence, group of pictures, picture, slice, macroblock, and block layers [18]. The video sequence layer contains one or more groups of pictures (GOPs). A GOP contains a sequence of pictures/frames beginning with an I picture followed by several P and B pictures. A picture corresponds to a single frame in the video sequence consisting of one or more 16-pixel high stripes called slices. A slice contains

a contiguous sequence of raster ordered macroblocks. Each macroblock contains a group of six $8 \times 8$ DCT blocks, four luminance blocks and two chrominance blocks. Each block corresponds to the basic coding unit on which the DCT is applied and consists of 64 pels arranged in an $8 \times 8$ array.

*3.2. MPEG-2 Video Coding.* The MPEG-2 coding standard [19] supports a wide range of bit rates, resolutions (both spatial and temporal), quality levels, and services for applications such as digital storage, High-Definition TV (HDTV), and so forth.

Unlike MPEG-1, MPEG-2 supports interlaced video input images which are scanned as even and odd fields to form frames. Thus, there are two new picture types. "Frame pictures" are obtained by interleaving the lines of an odd field and its corresponding even field while "field pictures" are formed from a field of pixels alone. All picture types can be I, P, or B frames. A coded I-frame consists of an I-frame picture, a pair of I-field pictures or an I-field picture followed by a P-field picture. A coded P-frame consists of a P-frame picture or a pair of P field pictures. A coded B-frame consists of a B-frame picture or a pair of B-field pictures.

MPEG-2 maintains MPEG-1 syntax, but uses extensions to add flexibility and functions. "Scalable" extensions support video data streams with multiple resolutions and the ability to partition the data stream into two pieces, one part containing all of the key headers, motion vectors, and low-frequency DCT coefficients and the second part transmitting less critical information such as high-frequency DCT coefficients. Other extensions offer temporal flexibility so not all frames have to be reconstructed.

*3.3. MPEG-4 Video Coding.* MPEG-4 was originally targeted to support low bit rates and error prone channels (e.g., for wireless devices) but also includes support for object-based user interactivity. MPEG-4 allows video objects to be placed anywhere in the coordinate system and transformations can be used to change the geometrical appearance of the objects. Streamed data can be applied to video objects to modify their attributes and the user's viewing point can be changed.

The basis of MPEG-4 video coding [20] is a block-based predictive differential video coding scheme as in MPEG-1 and MPEG-2. MPEG-4 video also specifies the coded representation of visual objects that can be synthetic (as in interactive graphics) or natural (as in digital TV). These visual objects can be combined to form compound objects. MPEG-4 multiplexes and synchronizes the visual objects before transmission to provide QoS and allows interaction with the scene generated at the receiver's end.

MPEG-4 video provides methods for compressing textures, for texture mapping of 2D and 3D meshes, compression of implicit 2D meshes, and compression of time-varying geometry streams that animate meshes. MPEG-4 also supports coding of video objects with spatial and temporal scalability. Scalability allows decoding a part of a stream and constructing images with reduced quality, reduced spatial resolution, reduced temporal resolution, or with equal temporal and spatial resolution but reduced quality.

*3.4. H.261 Video Coding.* H.261 [21] has many elements in common with MPEG-1. Both intraframe and interframe coding techniques are used for compression. Like MPEG-1, H.261 uses DCT-based frequency transformation and motion compensation (MC). The video is also organized in layers. However, there are some differences between these two coding standards. In H.261, the quantization is a single variable instead of a matrix of 64 terms and the syntax is simpler with only four layers. To minimize delay, only the previous picture is used for motion compensation. So, there are no B frames.

*3.5. H.263 Video Coding.* H.263 [22] is an evolutionary improvement to H.261, building on ideas from MPEG-1 and MPEG-2. H.263 is intended for low bitrate communication and it supports additional video frames. H.263 has MPEG-like blocks and macroblocks with prediction and motion compensation. The zigzagged quantized coefficients are coded using the MPEG run-level methods although with different tables. The video has four layers as in H.261.

Four optional modes enhance the functionality of H.263. The "unrestricted motion vector" mode allows motion vectors to point outside a picture. The "syntax-based arithmetic coding" mode supports arithmetic instead of huffman coding giving the same picture quality with fewer coded bits. The "advanced prediction" mode uses overlapped block motion compensation with four $8 \times 8$ block vectors instead of a single $16 \times 16$ macroblock motion vector. The "PB-frames" mode allows a P-frame and a B-frame to be coded together as a single PB-frame.

*3.6. H.264 Video Coding.* H.264 [23], MPEG-4 Part 10, or Advanced Video Coding (AVC) achieves very high-data compression. The goal of H.264/AVC was to provide good quality at substantially lower bit rates. An additional goal was to do this in a flexible way that would allow the standard to be applied to a wide variety of applications and to work well in a variety of networks and systems.

The basic functional elements (prediction, transform, quantization, and entropy encoding) are similar to previous standards. H.264 provides a number of new features that allow it to compress video much more effectively. These include what follows. (i) Multipicture motion compensation using up to 32 previously-encoded pictures as references. This usually allows modest improvements in bit rate and quality in most scenes. (ii) Variable block-size motion compensation (VBSMC) with block sizes as large as $16 \times 16$ and as small as $4 \times 4$, enabling very precise segmentation of moving regions. (iii) Quarter-pixel precision for motion compensation, enabling very precise description of the displacements of moving areas. (iv) A $4 \times 4$ integer block transform is used as opposed to the $8 \times 8$ DCT blocks. (v) Context-adaptive binary arithmetic coding (CABAC) is used to losslessly compress syntax elements in the video stream knowing the probabilities of syntax elements

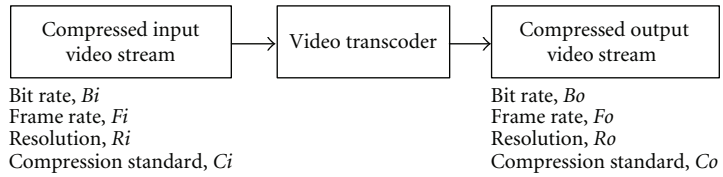


Figure 2: Video format conversion using a transcoder.

in a given context. (vi) Context-adaptive variable-length coding (CAVLC) a lower-complexity alternative to CABAC, is used for the coding of quantized transform coefficient values.

## 4. Video Adaptation Algorithms

To support the transmission of pre-encoded video over heterogeneous networks, the video streams need to be dynamically adapted based on the channel bandwidth and receivers' requirements. The device or system that performs this process is called a video transcoder. One salient function provided by video transcoding is bit rate conversion, which accepts a pre-encoded video stream as input and produces an output stream having a different bit rate. Other functionalities that may be provided by a transcoding process include conversion of the frame rate, spatial resolution, or compression standard (coding syntax). Figure 2 illustrates the transcoding process.

Different transcoding operations entail different levels of processing complexity. This complexity is determined by how much a compressed video stream must be decompressed before a transcoding operation is applied. Thus, a transcoding operation can be optimized by performing it in the appropriate stage of the compression/decompression process. Figure 3 shows different regions where various transcoding operations can be performed on a compressed discrete cosine transform and motion compensated video bit-stream. Region 1 represents uncompressed source image data where operations such as frame resizing (i.e., conversion of spatial resolution) and frame dropping (by reestimating motion vectors) can be performed relatively simply although a large amount of data has to be processed due to its uncompressed nature. Region 2 contains the same amount of data as region 1, but transcoding operations at this region can be performed without using the computationally intensive functions of Forward-DCT and Inverse-DCT transforms. Requantization (i.e., bit rate conversion) can be performed at this point by applying the new quantization factor on the dequantized DCT coefficients. In region 3, the data size is considerably smaller due to the absence of zero coefficients in DCT blocks that are quantized and run length encoded. Operations such as frequency filtering (i.e., bit rate conversion) and color to monochrome conversion are feasible at this region. Finally, region 4 contains fully compressed data and allows standard-specific and relatively simple operations such as intelligent frame dropping (i.e., frame rate conversion).

*4.1. Adaptive Requantization.* Requantization is an efficient transcoding technique for converting MPEG and H.261/263 video at a high bit rate to a lower bit rate. The requantization process involves several steps.

Figure 4 shows a requantization transcoder with bit rate control. First, the original video stream is decoded through variable length decoding (VLD) [24] to obtain the quantized DCT coefficients with coding information such as quantizer scale (also called quantizer step size), macroblock type, and motion vectors. A near-optimal decoding technique based on Huffman decoding is used to generate the quantized DCT values from the variable length codes in the compressed stream. An inverse quantizer then dequantizes these decoded coefficients using the quantization step size and produces the actual DCT coefficients. These coefficients are requantized with a larger quantization step size to reduce the bit rate. The quantized coefficients are then coded again with other coding information including the new quantization step and modified macroblock information through variable length coding (VLC) to get the resulting transcoded stream.

*4.1.1. Quantization.* The quantization process entails the division of the integer DCT coefficients by integer quantizing values. Intra- and interframe coding conform to different quantization rules. The quantized DCT coefficient (QDCT) in intracoding is calculated from the unquantized coefficient (DCT) by the following formulae [18]:

$$\mathrm{QDCT} = \frac{(16\times \mathrm{DCT}) + (\mathrm{Sign}(\mathrm{DCT})\times \mathrm{quantizer\_scale}\times Q)}{2\times \mathrm{quantizer\_scale}\times Q}, \tag{1}$$

where $Q$ is the quantization table value for the coefficient and the function Sign() in the rounding term produces the following values:

$$\mathrm{Sign}(\mathrm{DCT}) = \begin{cases} +1, & \text{when DCT} > 0, \\ 0, & \text{when DCT} = 0, \\ -1, & \text{when DCT} < 0. \end{cases} \tag{2}$$

For intercoding, a similar equation is used for the quantization with the exception that the rounding is always to the smaller integer value. Hence, the equation does not hold any rounding term:

$$\mathrm{QDCT} = \frac{(16\times \mathrm{DCT})}{2\times \mathrm{quantizer\_scale}\times Q}. \tag{3}$$

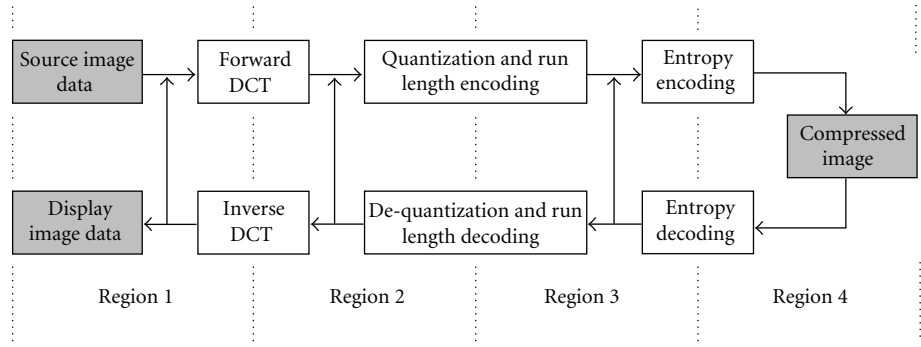


Figure 3: Different levels of compression/decompression process on a generic discrete cosine transform and motion compensation-based video.

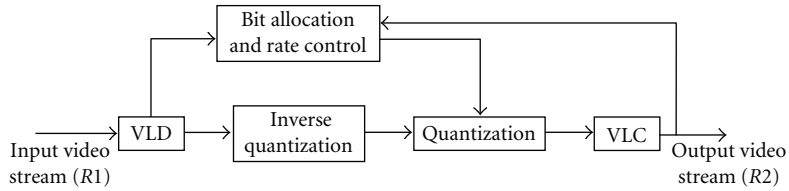


Figure 4: Transcoding using requantization.

*4.1.2. Dequantization.* Dequantization of the quantized DCT coefficients is performed by the inverse of the quantization procedure. For intra coding [18]

$$\text{DCT} = \frac{(2 \times \text{QDCT}) \times \text{quantizer\_scale} \times Q}{16}, \tag{4}$$

and for inter coding

$$\text{DCT} = \frac{((2\times\text{QDCT})+\text{Sign}(\text{QDCT}))\times\text{quantizer\_scale}\times Q}{16}. \tag{5}$$

*4.1.3. Rate Control.* Rate control in requantization is used to determine quantization parameters, and is responsible for preserving consistent video quality while satisfying both bandwidth and delay constraints. The relationship between quantizer step size and bit rate for a video stream can be used to determine the quantizer step size and bit allocation during requantization on a frame, slice, or macroblock basis. We used slice level rate control. The rate controller needs to know the target bit rate that is to be transmitted. At the slice level, the actual bit count in the original video stream can be scaled to obtain the target bit count. The scaling factor is the ratio between the transcoder's desired output (e.g., $R2$) and input bit rates (e.g., $R1$). This maintains the proportion of bits allocated among different frame types in the transcoded video sequence. The actual bit count from the original stream and corresponding target bit count for the $i$th slice in a frame are calculated as follows:

$$\begin{aligned} B_{\text{target}}(i) &= B_{\text{actual}}(i) \times \frac{R2}{R1}, \\ B_{\text{actual}}(i) &= B_{\text{stream}}(i) + \Delta, \end{aligned} \tag{6}$$

where $\Delta$ is defined as

$$\Delta = \begin{cases} 0, & \text{for the first slice of the first frame of any type,} \\ B_{\text{actual}}(i-1) - B_{\text{target}}(i-1), & \text{otherwise.} \end{cases} \tag{7}$$

To meet the target bit rate, the quantizer step size is adjusted based on feedback to the rate controller (as shown in Figure 4). The rate controller updates the quantizer step size for the next slice on the basis of the difference between the target and actual bit count for the previous slice (i.e., the value of $\Delta$). The new quantizer step size for the $i$th slice in a frame is calculated as follows:

$$Q(i) = Q_{\text{base}}(i) + \text{offset}, \tag{8}$$

where $Q_{\text{base}}(i)$ is the original quantizer step size for that slice and offset is determined by the value of $\Delta$ for the previous slice. For the first slice of each frame, offset is initialized to the mean value used for the previous frame of the same type.

As quantization is the only operation in the DCT-based video compression algorithm that introduces quality loss, requantization can produce some noticeable edge effects on the transcoded video stream. However, as each DCT coefficient is requantized to a smaller value, the bit rate reduction achieved by the mechanism is significant.

*4.2. Frame Dropping in Compressed Domain.* Frame dropping can be used to keep the generated bit rate of a video stream from exceeding the allocated channel bandwidth and to reduce the frame rate. Thus, a frame dropping

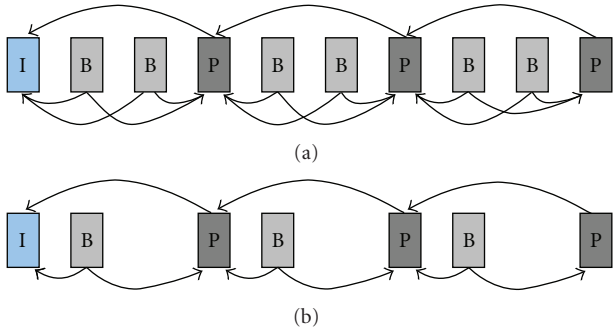


Figure 5: (a) A sample precoded MPEG-1 video sequence. (b) The transcoded sequence with alternate B frames dropped.

mechanism is used to reduce the data rate of a video stream in a sensible way by discarding a number of frames according to importance and transmitting the remaining frames at a lower rate. Usually, in a transcoding process, the video transcoder reuses the decoded motion vectors to speed up the re-encoding process [25]. In this case, the frames cannot be discarded because the motion vectors of each frame are estimated from their immediate predecessor frames. However, if frame dropping is allowed in a transcoding process, those motion vectors cannot be reused because the motion vectors of the current frame are no longer estimated from the immediate past frame. If frame dropping needs to be used, the current frame must be decompressed completely and the motion vectors have to be estimated again before recompression. This method introduces heavy computational overhead, which is undesirable in real-time transcoding. However, frames that are not referenced by other frames in the video sequence can be discarded in a specific interval to reduce the data rate thus avoiding the computation for recomputing motion vectors.

In an MPEG-1 video sequence, for example, I and P frames in a GOP are referenced by subsequent P and B frames in the group. Hence, I and P frames in the video sequence cannot be dropped without going through the process of re-estimation of motion vectors (done in region 1 of video compression/decompression processing shown in Figure 3) and thus requires complete decompression and recompression of the video sequence. On the other hand, B frames are not referenced by any other frames in the sequence. Therefore, a number of B frames in a specific interval can be discarded to control the bit rate while maintaining acceptable image quality. In this case, the transcoding operation is performed completely in the compressed domain (region 4 in Figure 3). By dropping a specific number of B frames, it is possible to produce a video stream with a desired rate, however, due to the small size of B frames in the video sequence, this approach has limited impact. A sample frame dropping scenario is illustrated in Figure 5.

*4.3. Frame Dropping in Pixel Domain.* To achieve increased data reduction, a pixel domain frame dropping technique

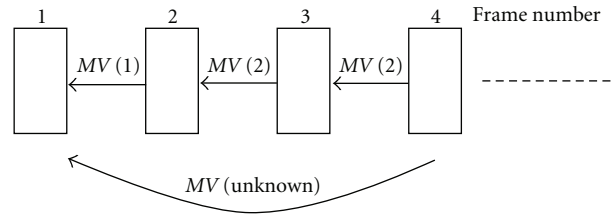


Figure 6: A motion tracing example.

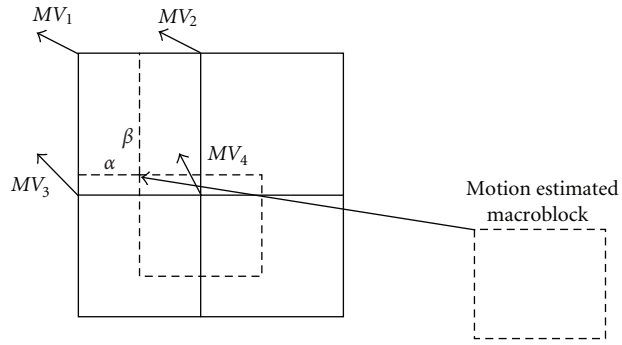


Figure 7: Interpolation of motion vectors.

could be used. In this case, the frame must be decompressed completely and motion estimation must be done again. To reduce the computational overhead of motion vector re-estimation, a bilinear interpolation method has been developed [26] to estimate the motion vectors for the current frame relative to the previous nondropped frame.

If the motion vectors between adjacent frames are known, the problem of tracing the motion from frame four to frame one as shown in Figure 6 could be partly solved using the repeated applications of bilinear interpolation. A shifted macroblock as shown in Figure 7 is located in the middle of four neighbor macroblocks. The bilinear interpolation is then defined as

$$MV_{\text{int}} = (1-\alpha)(1-\beta)MV_1 + (\alpha)(1-\beta)MV_2 + (1-\alpha)(\beta)MV_3 + (\alpha)(\beta)MV_4. \quad (9)$$

Here, $MV_1, \ldots, MV_4$ are the motion vectors of the four neighboring macroblocks. $\alpha$ and $\beta$ are determined by the pixel distance to $MV_1$. The weighting factor of each neighboring macroblock is inversely proportional to the pixel distance. By repeating the motion tracing, it is possible to create an extended motion vector for each macroblock in the current frame relative to its previously nondropped frame.

Bilinear interpolation only partially solves the motion vector reuse problem. Hence, further adjustment of the re-estimated motion vectors has to be performed by using a smaller search range. For each macroblock, the new position located by the interpolated and composed motion vectors is used as the search center for the final motion re-estimation stage.

The frame rate is controlled by dynamically determining the length of dropped frames. The goal is to make the

motion of the decoded sequence smoother. A threshold is set beforehand and if the accumulated magnitude of motion vectors after a nondropped frame exceeds this threshold, this frame is encoded. The threshold is determined by using the number of frames to divide the accumulated magnitude of motion vectors in a buffer. The threshold is recursively updated after transcoding each frame because the number of encoded frames should be dynamically adjusted according to the variation of the generated bits when the last nondropped frame is transcoded.

## 5. Low-Pass Filter

A low-pass filter provides rate reduction by progressively eliminating high-frequency components of the video signal [16]. In essence, the low-pass filter eliminates an appropriately determined number of DCT coefficients from the high-frequency ones that comprise a luminance or chrominance block. The low-pass parameter is related to the number of DCT coefficients left in each block after quality adjustment. At the beginning of each GOP, initial low-pass parameter values are set independently for I, P, and B pictures based on the compression ratio for the current GOP. The compression ratio for a GOP is calculated from the predicted size (in bits) of the GOP, the predictor for the total bits used by header data in the GOP, and the number of bits allowed for the current GOP, which in turn is calculated from a specified target rate, the number of pictures in the GOP, and the frame rate.

This technique implies that the rate averaged over a GOP-time is regulated by the target rate. However, the result of per-packet adjustment does not necessarily match the target rate. To make up the balance, an adjustment value is introduced. After the initial low-pass parameter value is set, it is changed dynamically for each of the following macroblocks in the GOP based on the size difference between the previous original and filter macroblock. Using this technique the low-pass parameter value for each macroblock is appropriately determined. By eliminating the specified number of DCT coefficients, it is possible to produce a video stream that has the desired rate.

## 6. Frame Resizing

One common solution to reduce bit rate is to generate a new compressed video with a lower spatial resolution from the original precoded video bit stream. This method requires downsizing of the original video and estimation of new motion vectors for each intercoded macroblock in the downsized video.

The algorithm proposed in [27] can achieve arbitrary image/video downsizing. This algorithm takes advantage of compressed domain processing techniques and is processed completely in DCT domains without introducing further computation. When combined with the transcoding method described in [28], which can estimate the motion vectors from the input bit stream for arbitrary downscaled video, the proposed method can efficiently process video stream downsizing.

In a spatial domain, for an arbitrary downsizing ratio $R$, defined as the ratio of original resolution to the desired resolution, more than one pixel in the original frame may contribute to one single pixel in the downsized frame [27]. For example, one $8 \times 8$ output block in the downsized frame can come from as many as $M \times N$ related blocks, which the supporting area in size of $8Rx \times 8Ry$ (where $Rx$ and $Ry$ are the horizontal and vertical downsizing ratio, resp.) may cover in the original frame. For arbitrary downsizing, these supporting areas may not align to the block border. This approach realizes downsizing in two steps: (1) extracting the supporting area from the original frame, and (2) downsizing it into an $8 \times 8$ output block.

Due to the noninteger downsizing ratio, some related blocks in the original frame might partially contribute to certain output blocks in the downsized frame. These related blocks can be totally covered or partially covered by the supporting area. The spatial information (DCT) of the supporting area is extracted by partially decoding the related blocks. The size of the extracted DCT block depends on the covered pixels of each related block. Then the supporting area can be represented by combining the extracted pixels from all the related blocks in DCT domain.

In a natural image, most of the signal energy is concentrated in the lower frequency part in the DCT domain. A reasonable downsizing scheme, as proposed in [29], is to retain only the lower frequency components and discard the high-frequency components of the block. Thus, most of the energy of the original block is preserved. As the supporting area of size $8Rx \times 8Ry$ may contribute to one $8 \times 8$ output block, it is necessary to discard the high-frequency component and extract only the low-frequency part of size $8 \times 8$ to downsize the block to $8 \times 8$ in the DCT domain.

The motion vector estimation approach proposed in [28] extends the existing video downsizing methods by considering an arbitrary downsizing scheme operating on several macroblocks. Since different numbers of pixels from the precoded macroblocks are used to form the new macroblock in the downsized video, existing methods using the spatial activity as a weighting factor for motion vector re-estimation are not well suited for the case of video downsizing by an arbitrary scale factor. The reason is that motion vectors are usually obtained by finding a matched macroblock within a search window of a reference frame by minimizing the sum of absolute differences (SAD) between the two macroblocks under comparison. To minimize the SAD, the new motion vector should be skewed toward the motion vector of the precoded macroblock that has more pixels involved in forming the new macroblock. For this reason, we consider another weighting factor, which is obtained by multiplying the number of horizontal pixels by the number of vertical pixels engaged from a precoded macroblock. Then the motion vector for each macroblock in the downsized video can be computed from the motion vectors and spatial activity (i.e., number of nonzero AC coefficients) of the related macroblocks of the precoded video, the weighting factor, and the downsizing factors.

## 7. Network Processors for Video Transcoding

To deliver enhanced next generation services such as converged voice and data, streaming video, and differentiated Quality of Service (QoS), an efficient in-network processing architecture needs to be developed. Such an architecture must be fast enough to process network data, flexible enough to allow for application specific functions and future upgrades, and reliable enough to provide QoS guarantees.

Network processors (NPs), specialized programmable CPUs optimized to perform packet processing at wire speed, can be used for this purpose. Network processors can perform many functions such as packet classification and possibly modification of packet contents at wire-speed near the edge of the network. The feasibility of network processor-based video transcoding is an area of research that has not yet been fully addressed.

## 8. Overview of Network Processors

Network processors have evolved over time to introduce greater processing and storage capability and, in some cases, additional functionality. These are critical issues in determining the feasibility of video transcoding using network processors.

*8.1. Intel IXP1200.* The IXP1200 contains one StrongARM processor core, six programmable multithreaded coprocessors (microengines), SRAM, SDRAM, and PCI and IX bus interface units.

The StrongARM 32-bit RISC microprocessor core running at 232 MHz is used for processing control packets, and doing tasks such as managing forwarding tables and other network state information. The microengines are minimal 32-bit RISC processor cores that are typically used to receive, process, and transmit packets independently. Each microengine supports four hardware threads so up to 24 threads can be executed "in parallel." The instruction store of each microengine has space for 1024, 32-bit instructions that each execute in one clock cycle.

The IX bus unit provides the on-chip scratchpad memory, receive and transmit queues (FIFOs), and a hash generation unit. The 64-bit IX bus connects the processor to Media Access Control (MAC) devices and is responsible for moving data to and from the receive and transmit FIFOs.

*8.2. Intel IXP2400.* The Intel IXP2400 [30] offers a wire-speed OC-48 data plane as well as control plane capability on a single chip. Each IXP2400 contains eight multithreaded packet-processing microengines, a low-power general-purpose Intel XScale core, network media and switch fabric interfaces, memory and PCI controllers, and interfaces to flash PROM and peripheral devices.

The eight 32-bit microengines run at 400/600 MHz and support multi-threading up to eight threads each. These microengines provide a variety of network processing functions in hardware. Each of the microengines has space for 4096, 40-bit instructions. The IXP2400 also offers Intel's Hyper Task chaining technology which allows a single stream packet/cell processing problem to be decomposed into multiple, sequential tasks that can be easily linked together. The hardware design uses fast and flexible sharing of data and event signals among threads and microengines to manage data-dependent operations among multiple parallel processing stages.

The integrated 32-bit XScale core offers high-performance processing of routing table maintenance and system management functions. The memory controllers facilitate efficient access to 32-bit SRAM and 64-bit DRAM, which hold the routing table, networking data, and so on. In addition, a programmable hash engine (48, 64, and 128 bit) is provided.

*8.3. Intel IXP2800.* The Intel IXP2800 [31] offers increased processing to support deep packet inspection and filtering, traffic management, and forwarding at up to OC-192 (10 Gbps) wire speed on a single chip. Its store-and-forward architecture combines a high-performance Intel XScale core with sixteen 32-bit independent multi-threaded microengines that cumulatively provide up to 25 Giga-operations per second.

The 32-bit Intel XScale core operates at up to 750 MHz. The sixteen 32-bit microengines running at up to 1.5 GHz, support multi-threading with up to eight threads each, and provide space for 8192, 32-bit instructions.

*8.4. IBM PowerNP.* The IBM PowerNP [32] supports multiple network interfaces including Gigabit Ethernet at 2.5 Gbps and OC-3 to OC-48 packet-over-SONET (POS). The core of the PowerNP contains 16 programmable protocol processing engines and seven coprocessors. Additional custom logic supports management of data movement at the physical and MAC layers.

The protocol processors are grouped into pairs which share a co-processor to accelerate packet processing. Each protocol processor supports two threads and includes a 3-stage pipeline (fetch, decode, and execute), general-purpose registers, eight instruction caches, and a dedicated ALU. The instruction memory (128 KB) consists of eight embedded RAMs and is initialized with picocode for packet processing and system management.

*8.5. EZChip (NP-1c).* The NP-1c [33] provides a scalable and programmable network processor architecture providing 10 Gbps wire speed. EZchip uses Task Optimized Processing Core (TOP core) technology. TOPs employ a super-pipelined and superscalar architecture for increased processing power. There are four types of TOPs, each having a customized instruction set and data path: (i) TOPparse identifies and extracts various packet headers and fields to classify packets; (ii) TOPsearch performs various table lookups required for layer 2 switching, layer 3 routing, layer 4 session switching, and layer 5–7 context switching and policy enforcement; (iii) TOPresolve allocates packets to an appropriate output port and queue; (iv) TOPmodify modifies packet contents.

Data plane packet processing in the TOPcore is pipelined; packets are passed from TOPparse to TOPmodify. A set of software commands from the system's host processor control the operations performed by the TOP processors. The programmability of the NP-1c makes it possible to adapt to new applications through simple changes in software without necessitating hardware changes.

*8.6. Agere (PayloadPlus).* The Agere PayloadPlus [34] network processor exploits pattern matching optimization technology to achieve high performance. Two main components are provided: the Fast Pattern Processor (FPP) and the Routing Switch Processor (RSP). The FPP is a pipelined, multi-threaded processor, that receives packets through the physical interface and carries out protocol recognition and classification at layers 2 through 7. The FPP reassembles traffic into protocol data units (PDUs) and sends them to the RSP which does queuing, packet modification, traffic shaping, and traffic management.

*8.7. Motorola (C-5e).* Motorola's C-5e [35] is capable of layer 2–7 processing at 5 Gbps. The C-5e contains 16 Channel Processors (CPs) for packet forwarding which each contain a transmit and receive Serial Data Processor (SDP) used for processing bit streams. The programmability of the SDPs supports diversity in media access control (MAC) interfaces, as well as parsing requirements, and can support different protocol implementations on a port-by-port basis. Each CP also contains a RISC core that is used for application-specific processing.

The Executive Processor (XP) integrated in the C-5e is responsible for supervisory tasks and management of the host processor. An on-chip Table Lookup Unit (TLU) offers a high-speed flexible classification engine that supports over 46 million IPv4 lookups per second. The TLU is connected to a 64-bit 128 MB SRAM. The C-5e also contains 128 MB SDRAM for payload storage. By connecting multiple C-5e NPs through their fabric interfaces to a fabric switch, it is possible to achieve Terabits per second of aggregate bandwidth. The C-5e NP's highly configurable Fabric Processor (FP) enables implementation of per-flow congestion control, segmentation and re-assembly, and integrated scheduling of up to 128 queues.

## 9. Implementation of the Active Video Adaptation Node

We used the Intel IXP1200 network processor to implement our active video adaptation node [36]. More powerful, second generation, NPs suggest that still better results than those we report are now achievable. The architecture of our active video adaptation node conforms to the node architecture provided by the active networks "reference model" [8]. An active node runs a Node Operating System (NodeOS) and one or more Execution Environments (EEs) and provides services to users through Active Applications (AAs). The functionality of an active network node is divided among these three major components.

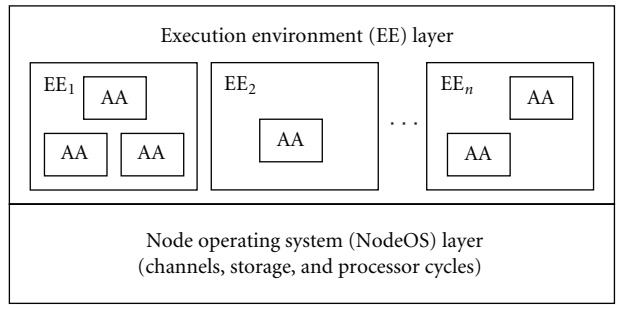


Figure 8: Schematic representation of the active node architecture.

*9.1. Active Node Components.* Figure 8 illustrates the architecture of an active network node. Underlying each active network node is a Node Operating System that manages the resources of the active node such as processors, channels, and memory. The channels implemented by the NodeOS carry packets to and from underlying communication substrates and also perform protocol processing. An Execution Environment provides a virtual machine programming interface for executing programs on the active node. Thus, an EE is analogous to a shell program in a general purpose computing system exporting, in this case, an interface through which end-to-end network services can be implemented. Multiple EEs can be present on a single active node at the same time. EEs are isolated from the details of resource management by the NodeOS. An EE can accept active packets that initiate the execution of packet specific programs, also called Active Applications. AAs program the virtual machine provided by an EE to implement an end-to-end service. The code constituting the AA may be contained in a packet itself or, more likely, preloaded at the node. An EE can invoke multiple AAs to provide multiple services simultaneously and manages the initiation, execution, and termination of these AAs. All of the EE and AA functionalities are programs running on the microengines and the NodeOS functionalities are provided by the operating system running on the StrongARM.

We have implemented our video adaptation algorithm as an AA. The video adaptation AA is notified of a target output rate for the input video stream by the EE that initiates it. For the adaptation of multiple independent video streams belonging to different multicast groups, multiple AAs may be initiated by the same or different EEs. The same is also true for the adaptation of a single video stream into streams having different data rates. Figure 9 illustrates these two scenarios.

*9.2. Flow of Video Packets through the Active Node.* Figure 10 shows a general flow of video packets through the active node. Once video packets are received on an input port they are classified based on information contained in the packet headers such as protocol number, port numbers in UDP headers, and/or Type ID in an ANEP [37] header. Incoming packets may have to wait in one or more queues before

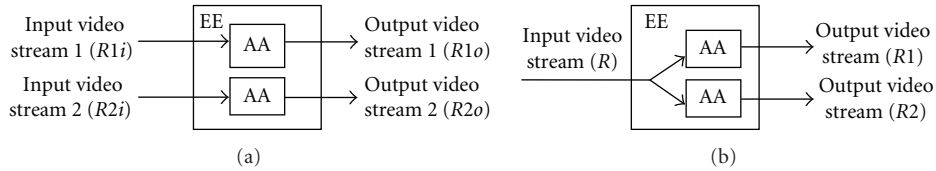


Figure 9: (a) Two video streams with rates $R1i$ and $R2i$ are adapted to streams with rate $R1o$ and $R2o$, (b) A single stream with rate $R$ is adapted to two different streams with rate $R1$ and $R2$.

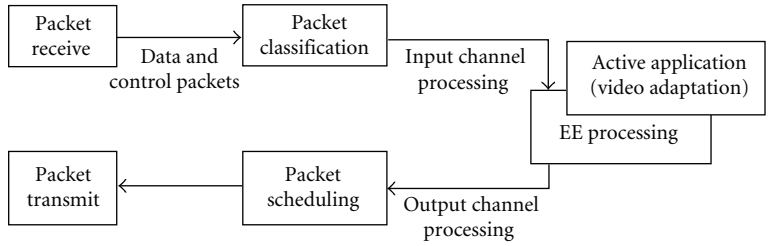


Figure 10: Video packet flow through the active node.

classification. The classification of packets determines the input channel to the appropriate EE to which packets are directed. Afterwards, the input channel processing packets are passed on to the corresponding EE. Upon receiving a packet, the EE sends it to the appropriate AA and receives the result packet from the AA after adaptation of the packet in accordance with the specified target rate. Video adaptation AAs used by the EE are chosen based on a set of identifiers consisting of a source IP address and port number, a multicast group address for sending the video packet, and a destination port number. On the output side, the EE sends the adapted video packets to the scheduler through output channels. Packets are then transmitted through appropriate output ports. Before transmission, the packets may have to wait in output queues. Besides active video packets having an ANEP header, the node can also process legacy traffic (conventional IP packets) by setting up the appropriate channels that simply forward the packets without applying any video adaptation.

*9.3. Mapping the Algorithms to the IXP1200.* Reception and classification of packets, transcoding, and scheduling and transmitting adapted packets are all implemented on the microengines to provide wire-speed packet processing. The processing done by each microengine differs and is determined prior to run time.

*9.3.1. Receiving Packets.* We allocate microengine zero for receiving packets. All four hardware threads are used for this task. Incoming packets received by microengine zero are queued to avoid packet loss during packet processing by other microengines. Each thread on the receiving microengine queues packets to be used by the microengines that classify, transcode, and finally retransmit them. We have implemented array-based circular queues in SRAM where each entry in the queue contains a packet descriptor consisting of the packet buffer handle (in SRAM) and the packet size. We experimented with two different queue configurations: single input and multiple input.

*Single Input-Queue Configuration.* In this configuration, one 100 Mbps port is mapped to a single packet queue to be served by all four threads of the classification microengine. Each of the four threads in the receiving microengine is dedicated to receiving packets from the single port and to queuing them. The single queue implementation can be used for the adaptation of a single input video stream into single or multiple output streams having different data rates. Figure 11 shows the single input-queue configuration.

*Multiple Input-Queue Configuration.* In this configuration a queue is created for each of the four threads in the receiving microengine. The assignment of these receiving threads to input ports can be done in different ways. Each thread can be assigned to a single port or to a different port (specified in advance). Thus, the receiving microengine can receive packets from up to four ports. The multiple-queue implementation can be used for transcoding single or multiple video streams. In single stream transcoding, packets received by the four threads are put in the queues sequentially. In multiple-stream transcoding, each queue contains packets belonging to a separate video stream that is handled by a thread assigned to that queue. The multiple-queue configuration is illustrated in Figure 12.

*9.4. Classification of Packets.* We allocate microengine one for dequeuing packets from the input queues and classifying them. Packet classification separates convention IP data packets from the video packets to be transcoded. Both dequeuing and classification of a given packet are performed by a single thread. Packets are also analyzed to extract MPEG-1 video start codes. In the single queue configuration, each thread waits for its turn to retrieve a packet from the queue.

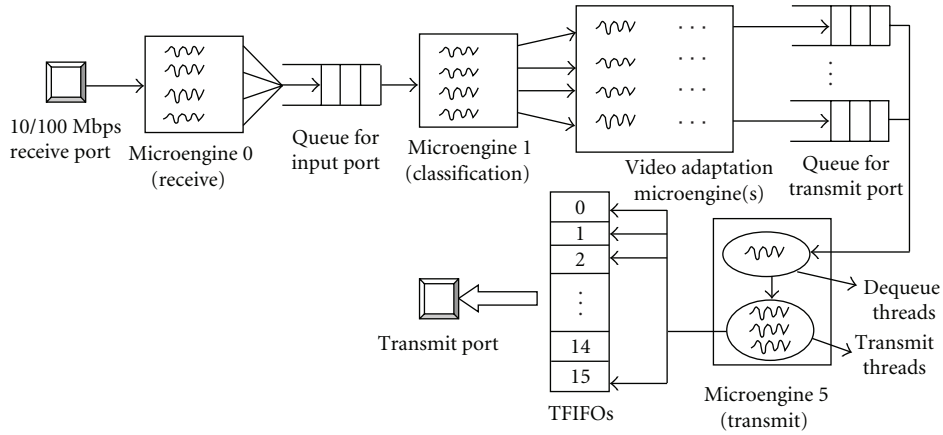


Figure 11: Single queue configuration for the video transcoding node.

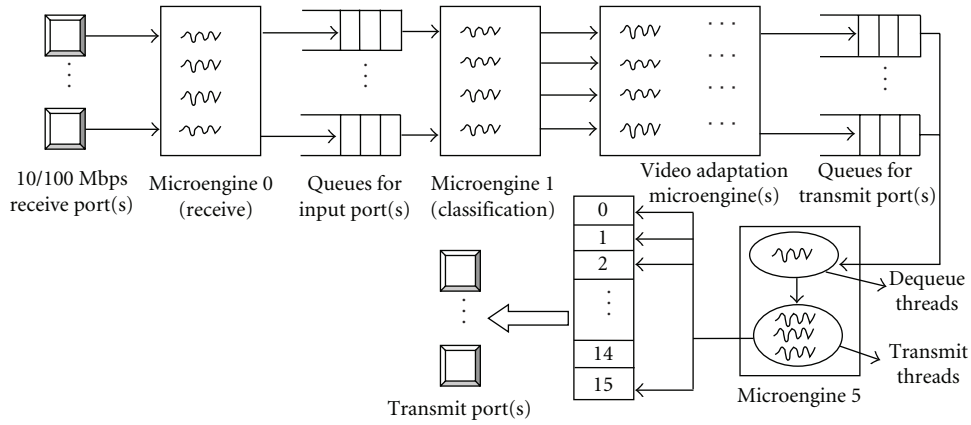


Figure 12: Multiple queue configuration for the video transcoding node.

In the multiple-queue configuration, threads wait for new entries in their respective queues.

*9.5. Video Transcoding.* We allocate three microengines (2, 3, and 4) for transcoding video packets from the classification microengine. All three microengines are used by the requantization technique while one microengine is sufficient for implementing the frame dropping technique. Video transcoding for a packet is done by a single thread in each of these video processing microengines, as shown in Figures 11 and 12.

*9.5.1. Requantization.* Processing DCT coefficients during requantization requires a simple but frequent operation due to the large portion of MPEG-1 video data coming from the block layer consisting of these coefficients. Implementing the entire requantization process in one microengine is not possible since each microengine has a limited instruction store capable of holding only 1024 instructions. Hence, we allocate microengine 2 for processing up to the macroblock layer and microengines 3 and 4 for processing the block layer.

Packet data are fed into the input buffer from the SDRAM. Microengine 2 processes the data from the sequence layer to the macroblock layer and moves the necessary coding information to the shared SRAM and SDRAM for processing the block layer on microengines 3 and 4. The picture and slice layers are parsed to extract the frame type (I, P, or B) and quantization scale, respectively. The frame type determines the macroblock coding type (i.e., intra- or intercoding) which defines the dequantization approach required (i.e., intraframe or interframe). The quantization scale is used to dequantize the original DCT coefficients. The coding information includes the macroblock type and quantization scale for processing the DCT coefficients. Microengine 2 also processes the macroblock layer based on the macroblock type to obtain the macroblock pattern, motion vectors, and macroblock quantization scale. The macroblock pattern and quantization scale (if it is different from the slice layer quantization scale) are also included in the coding information. The macroblock pattern provides a coded block pattern that describes which blocks within the macroblock are coded.

The block layer processing is divided into two parts. The DC components of the block are processed by microengine

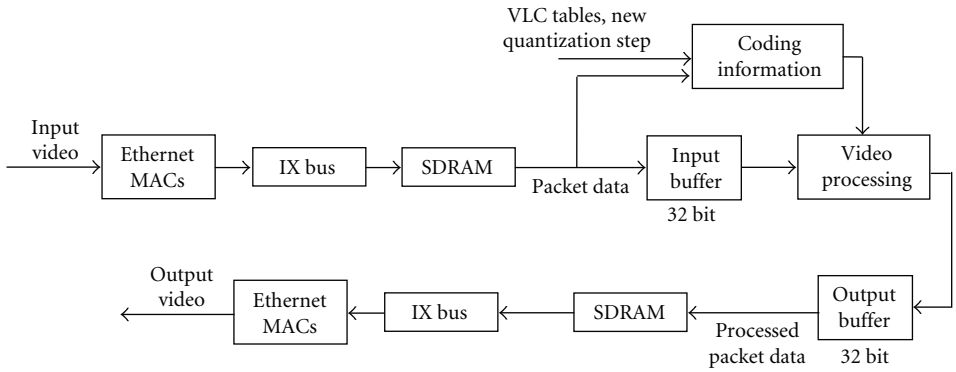


Figure 13: Data flow through the input and output buffers during requantization.

3 and the AC components are processed by microengine 4. The microengine threads obtain the coding information from the SRAM and SDRAM and additional information such as the quantization table (intra or inter), the VLC tables, and a new quantization scale (chosen according to the rate control mechanism described in the previous section) from the IXP 1200's scratchpad memory. The microengines decode the quantized DCT coefficients and dequantize them to get the actual DCT coefficients. The coefficients are then requantized with the new quantization step and are encoded using VLC to get the transcoded DCT coefficients. The microengines store the processed DCT coefficients in the output buffer and write them back to the SDRAM. Figure 13 illustrates the data processing through the input and output buffers during requantization. Once the whole packet has been processed, the thread puts it in the correct output queue (one for each of the four threads in microengine 4). The transmit microengine dequeues the packets from the output queues in round robin fashion and transmits them through one or more output ports.

*9.5.2. Frame Dropping.* Frame dropping refers to a compressed domain transcoding technique which is relatively simple and entails less video computation. We allocate one microengine (number 2) for transcoding the video stream through frame dropping. As the transcoding technique selectively drops a number of B frames to produce an output video stream with the desired rate, the microengine processes the video data from the sequence layer to the picture layer to recognize the frame type. The picture rate in the sequence layer is updated based on the new frame transmission rate. The microengine achieves the new transmission rate by discarding a specific number of B frames. The microengine threads put the packets that belong to a nondropped frame into the output queues. A single queue is created for each thread in the microengine. Like requantization, packets are dequeued from the output queues in a circular manner and are transmitted through appropriate output port(s).

*9.6. Transmitting Packets.* We allocate microengine 5 for dequeuing packets from the output queues and transmitting them. One thread is assigned to dequeue the packets and the remaining threads are used to schedule and send the packets out on the wire as illustrated in Figures 11 and 12. Once a packet is received from the output queue, new checksums are calculated for the IP and UDP headers based on the new packet size and contents. Packets are then transmitted out of the IXP1200.

*9.7. Hashing.* The IXP1200 provides a hardware hash unit located within the IX bus unit and the hash unit implements a hash function that produces values with a uniform statistical distribution regardless of the input. Packet processing that requires one or more table lookups is greatly eased by the hardware assisted hashing functionality provided by the IXP1200. This hashing unit is only accessible to microengines and is capable of performing either 48 bit or 64 bit hashes.

In our framework, we have used a hash table to store, retrieve, and update hints from the receivers at the active node. These hints describe user preferences such as interest levels in particular video streams and the capability of the end devices such as processing power, screen size, and so on. Receivers transmit the hints upstream (through the active node) to the sender in capsule form. Video packets originating from the source pass through the active node that adapts the packets based on the hints from the intended receivers and network connections.

*9.8. Implementation Complexity and Portability.* The active video adaptation node entails greater implementation complexity than a conventional router to provide support for video adaptation. This complexity and the associated overhead is not insignificant. Fortunately, the "cost" of designing the adaptation node is incurred only once and may be amortized over the number of network processors it runs on. Further, the programmability of NPs will allow for relatively easy extension of adaptation node functionality. Despite the high-level similarity of NP architectures, there is currently no readily accepted standard programming model for network processors. Thus, our code is largely IXP specific.

The different phases of the video adaptation algorithm executed by each microengine are predetermined prior to

execution. This makes the allocation of work simpler than assigning it to the microengines dynamically. The concept of "microblock" in packet receiving is used to balance the code performance and modularity. A microblock is a sequence of microengine code that operates on a single packet that is currently being processed. One or more microblocks can be combined together in a loop, called a "dispatch" loop that calls these microblocks in a particular order to perform specific functions. We have used a while loop as the dispatch loop that calls the packet receiving microblock and a packet queuing microblock to receive packets from the input ports and to queue them for use by the other microengines. As the packet length is unknown during the reception of the start of the packet, a fixed size buffer (large enough to hold the maximum size packet) is allocated for each packet. Hardware supported SRAM LIFO stacks are used for such buffer allocation without any overhead for writing code to create stacks. This approach is largely generalizable to most current NP architectures.

Another challenge was to determine the granularity of segmentation for an MPEG-1 video stream at the server. We decided to segment the video stream into a sequence of independent packets on a per slice basis since the slice in a frame represents the smallest independent coding unit in an MPEG-1 video sequence. This suited the comparatively low power of the IXP1200 but could still be used with more powerful NPs. The granularity of segmentation does not significantly affect coding complexity.

We used a nonpreemptive thread arbiter and shared variables to implement intra-microengine thread communication and mutual exclusion. The nonpreemptive thread arbiter allows one thread to run until the thread itself explicitly releases control of the microengine by waiting for a signal, swapping out on a memory read, or voluntarily releasing control. Shared variables are implemented in the hardware with absolute registers providing extremely fast interthread communication and mutual exclusion. Synchronization was a key performance issue in our IXP 1200 implementation and will likely continue to be regardless of network processor architecture.

Most of the computation overhead and complexity comes from implementing different levels of decompression and compression tasks during the actual transcoding of the video packets. However, multiple microengines capable of wire-speed packet processing made the transcoding task fast enough to provide a realistic video transfer rate (24–30 frames per second for MPEG-1) despite the limitations of the IXP1200. Processing requiring table lookups was simplified and expedited by the hardware assisted hashing functionality provided by the IXP1200.

Implementing our video adaptation algorithm on other network processors would require a modified strategy for allocating various tasks such as reception and classification of packets, transcoding, and scheduling and transmitting adapted packets based on the hardware resources provided by the particular network processor. Similarly, other architecture specific coding patterns would have to be revised. Not surprisingly, porting NP code is similar to porting low-level parallel code.

## 10. Protoype Evaluation

We now present the results obtained from our evaluation of our IXP1200-based video transcoder. The metrics used to measure transcoder performance included transcoding latency, throughput, and accuracy. These criteria capture the effectiveness of transcoding MPEG-1 streams using the IXP1200. Transcoding latency determines how fast video transformation takes place. Low transcoding latency is preferred. The goodness of transcoding latency is determined by how close the transcoded video transfer rate is to the actual video transfer rate (i.e., for MPEG-1, 24–30 frames per second). Transcoding throughput determines the number of streams that can be processed simultaneously. Finally, the amount of "blockiness," "blurriness," and "noise" in the transcoded video determines the accuracy of transcoding which is assessed by comparing the quality of the transcoded streams with that of the original streams.

A test environment was set up to verify the practicality and applicability of the video transcoding mechanism. The experimental system consisted of a video server (sender), an IXP1200-based transcoding node, and several clients connected to the trancoding node. A number of experiments were conducted to transcode MPEG-1 video streams according to the various requirements of the clients and their link bandwidths.

For our experiments, we used a selection of MPEG-1 video streams whose properties are summarized in Table 1. The frame size represents the width and height of each frame in pixels. The average size of I, P, and B frames are given in bytes. The frame pattern represents the order in which frames in a GOP are displayed. Each of the video streams is viewed at a rate of 30 fps.

*10.1. Transcoding Latency.* We experimented using various numbers of microengine threads to get an idea of the capabilities of the IXP1200. The test video streams were transmitted from the video server to the video transcoding node at a rate of 300 Kbps. We set the output target rate of the active node to 200 Kbps. We then conducted experiments for each video sequence to observe any packet loss. The system was found to be capable of processing the video streams at that input rate.

*10.1.1. Requantization.* Figure 14 and Table 2 show the transcoding latency per frame for each test video stream using requantization. The transcoding latency for requantization includes the time required for decoding the frame through VLD, further processing related to dequantization and quantization of DCT coefficients, and encoding using VLC.

To take advantage of the IXP1200's multithreading architecture and to improve transcoding performance, multiple threads in each microengine can be used. The latencies (approximate values in milliseconds) are shown in Table 2 and Figure 14 for different numbers of threads running on the transcoding microengines. Employing two threads leads to lower transcoding latency and the same is true for higher

Table 1: Properties of the test MPEG-1 video sequences.

| Clip Name | Frame Size | I | P | B | Frame Pattern |
|---|---|---|---|---|---|
| ski.mpeg | 192 × 144 | 2877 | 1631 | 1012 | IBBPBBPBBP |
| danger.mpeg | 240 × 176 | 3586 | 1959 | | IPIIPI |
| canyon.mpeg | 144 × 112 | 2265 | 1881 | 445 | IBBPBBIBBP |
| vessel.mpeg | 256 × 256 | 4567 | 4844 | 4345 | IBBPBBIBBP |
| blazer.mpeg | 128 × 96 | 2431 | 1419 | 567 | IBBPBBPBBP |
| rotate.mpeg | 160 × 112 | 2670 | | | I |

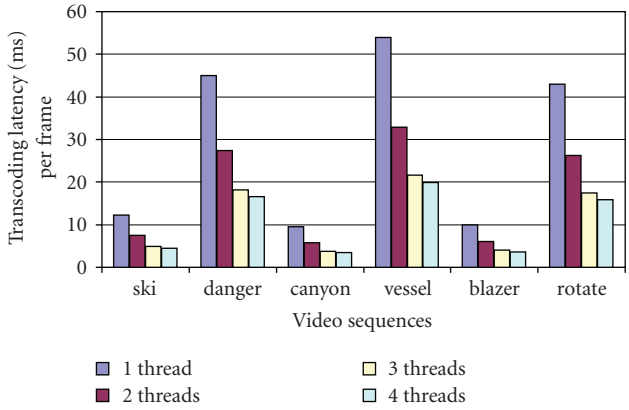


Figure 14: Transcoding latency per frame using requantization.

numbers of threads. However, employing more than three threads provides little benefit. The reason for this is the increasing level of intramicroengine mutual exclusion that is required.

The transcoding latencies shown in Table 2 indicate that the test video sequences can be transcoded at 24–30 frames or more per second while using two or more threads in each video transcoding microengine. This meets the minimum display frame rate for MPEG-1 video. Note that a single thread implementation cannot meet this transcoding rate for all the test video sequences. (e.g., "danger.mpeg," "vessel.mpeg," and "rotate.mpeg" are transcoded at rates of around 22.22, 18.51, and 23.28 fps which are at least close to the minimum display rate, 24 fps.) The salient point to note from Table 2 (with reference to Table 1) is the relationship between frame size and transcoding latency. The smaller frame sizes or coding bit rates imply sparser blocks in the frame resulting in greater speedup during requantization. The opposite is true for larger frames.

In our next experiment, we evaluated the transcoding latency for the transcoding of two video streams (e.g., intended for different multicast groups). (No actual multicast implementation was done. For assessing performance of transcoding multiple streams, however, this was not required.) Two threads (dedicated) in each video transcoding microengine were responsible for transcoding one video stream. The transcoding latency obtained for each video stream was slightly higher than (previous experiment) when they are separately transcoded with two threads in each microengine. For the streams "canyon.mpeg" and "rotate.mpeg," the latencies obtained per frame were 7.80 and 30.64 milliseconds, respectively, which are slightly higher than in the case of the single stream transcoding (6.41 and 26.18 milliseconds from Table 2). This is due to intra-microengine mutual exclusion.

*10.1.2. Frame Dropping.* Table 3 and Figure 15 show the transcoding latency (approximate values in microseconds) per frame for several test video streams that contain B frames in the sequence. The major component of the transcoding latency using frame dropping includes the time required to process the video data from the sequence to the picture layer and the slice headers of a frame.

As in requantization, employing two threads leads to lower transcoding latency than using one thread and the same is true for higher number of threads. However, as shown in Table 3 and Figure 15, employing more than two threads contributes little to the performance. This is due to the greater amount of intra-microengine mutual exclusion that is performed by the video adaptation code.

Transcoding latencies obtained for frame dropping indicate the real-time performance of this compressed domain transcoding technique. Compared to requantization, transcoding speed is much higher in this case. This is due to the amount of video data that must be processed by the frame dropping code. Approximately 4.83% of video data coded at 333 Kbps, as an example, come from the sequence to the picture layer and the slice headers of a frame.

Since the frame dropping technique does not process macroblock and block layer data, the transcoding latency for a frame depends on the number of slices in the frame. As each frame is processed per slice, more slices result in more packets that require increased processing time. The opposite is true for a smaller number of slices per frame. This is reflected in Table 3.

In this case also, we evaluated the latency for the transcoding of two video streams intended for different multicast groups. Two threads in each video transcoding microengine were responsible for transcoding one video stream. The transcoding latency obtained for each video stream was slightly higher than (previous experiment) when they are separately transcoded with two threads in each microengine. For the streams "ski.mpeg" and "vessel.mpeg," the latencies obtained per frame were 346 and 418 microseconds, respectively, which are a little higher than the values obtained in the case of the single stream transcoding (335 and 403 microseconds from Table 3).

Table 2: Transcoding latency per frame using requantization.

| Clip Name | Transcoding Latency (ms) | | | |
|---|---|---|---|---|
| | 1 Thread | 2 Threads | 3 Threads | 4 Threads |
| ski.mpeg | 12.21 | 7.44 | 4.96 | 4.51 |
| danger.mpeg | 45.00 | 27.41 | 18.22 | 16.61 |
| canyon.mpeg | 9.52 | 6.41 | 4.36 | 4.06 |
| vessel.mpeg | 54.00 | 32.90 | 21.67 | 19.90 |
| blazer.mpeg | 9.88 | 6.52 | 4.56 | 4.19 |
| rotate.mpeg | 42.95 | 26.18 | 17.45 | 15.87 |

Table 3: Transcoding latency per frame using frame dropping.

| Clip Name | Slices | Transcoding Latency ($\mu$s) | | | |
|---|---|---|---|---|---|
| | | 1 Thread | 2 Threads | 3 Threads | 4 Threads |
| ski.mpeg | 11 | 450 | 335 | 290 | 281 |
| canyon.mpeg | 7 | 345 | 248 | 219 | 214 |
| vessel.mpeg | 15 | 590 | 403 | 341 | 338 |
| blazer.mpeg | 8 | 388 | 268 | 237 | 231 |

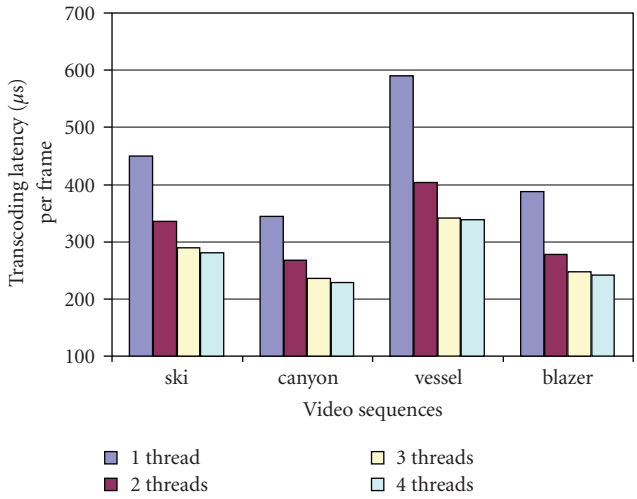


Figure 15: Transcoding latency per frame using frame dropping.

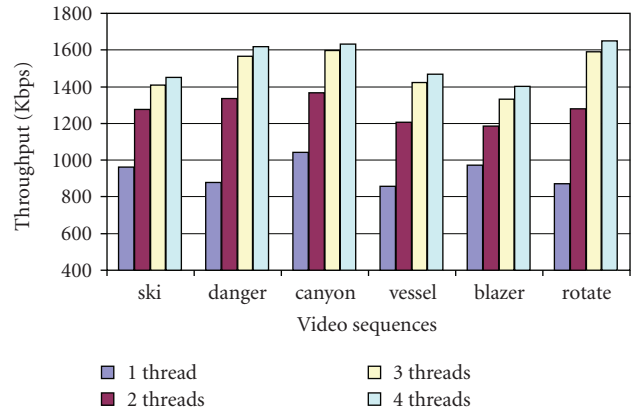


Figure 16: Throughput using requantization.

*10.2. Throughput.* Next, we measured and compared the throughput attained by the active video adaptation node for the requantization and frame dropping techniques.

*10.2.1. Requantization.* For our throughput experiments, we set a target of 30% reduction in data rate for the output streams. The input rate of the test video streams from the video server to the transcoding node was then gradually increased until the node experienced packet loss. Thus, the throughput for each video stream was measured using the highest possible injection rate at which the node could perform the transcoding without losing packets.

Throughputs afforded (approximate values in Kbps) are shown in Table 4 and Figure 16 for different numbers of threads running on the video transcoding microengines. Using two threads leads to higher throughput than using one thread and the same is true for higher numbers of threads. However, employing more than three threads has decreasing significance. The reason is, again, the amount of intra-microengine mutual exclusion. Overall, requantization produces a significant reduction in data rate while maintaining reasonable image quality (as discussed later), but at the expense of incurred delay. The results from Table 4 show that requantizing streams with high transcoding latency results in decreased throughput.

We also evaluated throughput when transcoding two video streams targeted for different multicast groups. The total throughput attained for these two streams was somewhat lower than the overall throughput attained when they were separately transcoded using two threads in each microengine. For example, the throughput obtained from this experiment for the video streams "ski.mpeg" and "vessel.mpeg," 2214 Kbps, was a little lower than the aggregate throughput obtained in the case of the single stream implementation (1277 + 1208 = 2485 Kbps from Table 4).

Table 4: Throughput of video transcoding for MPEG-1 sequences.

| Clip Name | Throughput (Kbps) | | | |
|---|---|---|---|---|
| | 1 Thread | 2 Threads | 3 Threads | 4 Threads |
| ski.mpeg | 962 | 1277 | 1409 | 1450 |
| danger.mpeg | 880 | 1335 | 1565 | 1620 |
| canyon.mpeg | 1044 | 1366 | 1596 | 1634 |
| vessel.mpeg | 859 | 1208 | 1423 | 1470 |
| blazer.mpeg | 973 | 1185 | 1333 | 1402 |
| rotate.mpeg | 870 | 1279 | 1591 | 1650 |

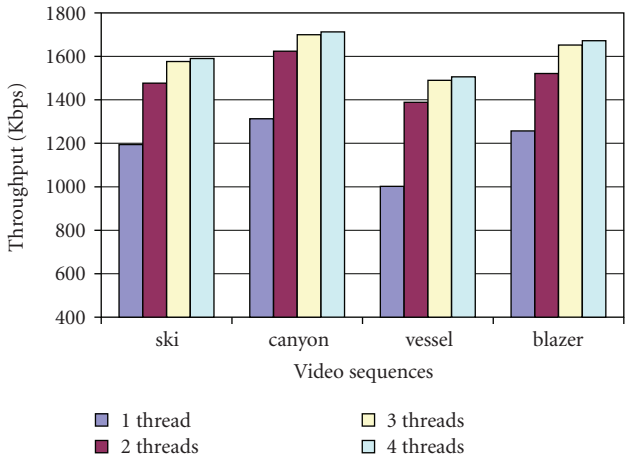


Figure 17: Throughput using frame dropping.

*10.2.2. Frame Dropping.* As in requantization, the input rate of the test video streams from the video server to the active node was gradually increased until the node experienced packet loss. Since the test video sequences were coded at 30 frames per second, we decided to drop six B frames from every 30 consecutive frames to meet the minimum requirement of MPEG-1 picture rate (24 frames per second). Table 5 and Figure 17 show the throughput attained for the test video streams that contain B frames.

The throughputs (approximate values in Kbps) are shown for changing numbers of threads running on the video adaptation microengines. The trend in the change of throughput for frame dropping based on the change in the number of microengine threads is similar to that for requantization. As in our compressed domain frame dropping, B frames are selectively dropped, the reduction in output data rate depends on the number and size of the B frames in the video sequence. A video stream having high average size of B frames produces less post frame dropping throughput. This is illustrated in Table 5 for the sequence "vessel.mpeg."

We also evaluated throughput when transcoding two video streams targeted for different multicast groups. The total throughput attained for these two streams was, again, a little lower than the overall throughput attained when they were separately transcoded using two threads in each microengine. For example, the throughput obtained from this experiment for video streams "blazer.mpeg" and "vessel.mpeg," 2657 Kbps, was slightly lower than the total throughput obtained in the case of the single stream implementation (1389 + 1521 = 2910 Kbps from Table 5).

Compared to requantization, frame dropping produces less reduction in data rate. Moreover, the reduction in data rate achieved by the frame dropping technique is limited by the number and size of B frames in the video sequence. For this reason, this compression domain frame dropping technique may not always be able to meet the data rate desired by the receiver. However, this transcoding technique is well suited to be used for ensuring that a receiver receives frames at a rate appropriate to its processing capabilities. On the other hand, requantization produces larger reduction in data rate though it involves more processing and thus takes longer than the other technique. Unlike the frame dropping technique, requantization reduces the data rate without affecting the frame rate, but of course at the cost image quality.

*10.3. Accuracy of Video Adaptation.* We also measured the quality of the transcoded video streams by comparing them with the original streams. In evaluating the transcoded video quality, we made use of peak signal-to-noise ratio (PSNR) and percentage error (PE) introduced, as our error metrics. The PSNR measures are estimates of the quality of a transcoded image compared to the corresponding original image. PSNR in decibels (dB) is computed as

$$\text{PSNR} = 20\log_{10}\left(\frac{255}{\text{RMSE}}\right) \tag{10}$$

for an 8-bit (pixel values of 0 to 255) image where RMSE represents the root-mean-squared error of the transcoded image. For an original image $F_o(i, j)$ of $N$ by $N$ pixels and the corresponding transcoded image $F_t(i, j)$ of the same size, RMSE is computed as

$$\text{RMSE} = \frac{\sqrt{\sum_{i=1}^{N}\sum_{j=1}^{N}\left[F_t(i,j) - F_o(i,j)\right]^2}}{N}. \tag{11}$$

Next, the PE introduced in a transcoded image is calculated from the relative error (RE) as follows:

$$\text{RE} = \frac{\sum_{i=1}^{N}\sum_{j=1}^{N}\left(\left|F_t(i,j) - F_o(i,j)\right|\right)}{\sum_{i=1}^{N}\sum_{j=1}^{N}F_o(i,j)}, \tag{12}$$

$$\text{PE} = \text{RE} \times 100\%.$$

Table 5: Throughput of video adaptation for different test MPEG-1 sequences using frame dropping.

| Clip Name | Throughput (Kbps) | | | |
|---|---|---|---|---|
| | 1 Thread | 2 Threads | 3 Threads | 4 Threads |
| ski.mpeg | 1194 | 1477 | 1576 | 1590 |
| canyon.mpeg | 1313 | 1624 | 1700 | 1713 |
| vessel.mpeg | 1002 | 1389 | 1490 | 1506 |
| blazer.mpeg | 1257 | 1521 | 1652 | 1669 |

Table 6: Quality of various test MPEG-1 video sequences-PSNR and PE.

| Clip Name | I frame | | P frame | | B frame | |
|---|---|---|---|---|---|---|
| | PSNR | PE | PSNR | PE | PSNR | PE |
| ski.mpeg | 40.92 | 0.51 | 33.26 | 0.72 | 33.88 | 0.70 |
| canyon.mpeg | 28.57 | 6.56 | 28.64 | 6.53 | 28.27 | 6.90 |
| danger.mpeg | 26.60 | 7.34 | 22.98 | 11.38 | | |
| vessel.mpeg | 24.83 | 2.32 | 23.99 | 2.50 | 26.57 | 1.96 |
| blazer.mpeg | 29.02 | 2.06 | 28.57 | 2.07 | 28.30 | 1.97 |
| rotate.mpeg | 22.90 | 6.52 | | | | |

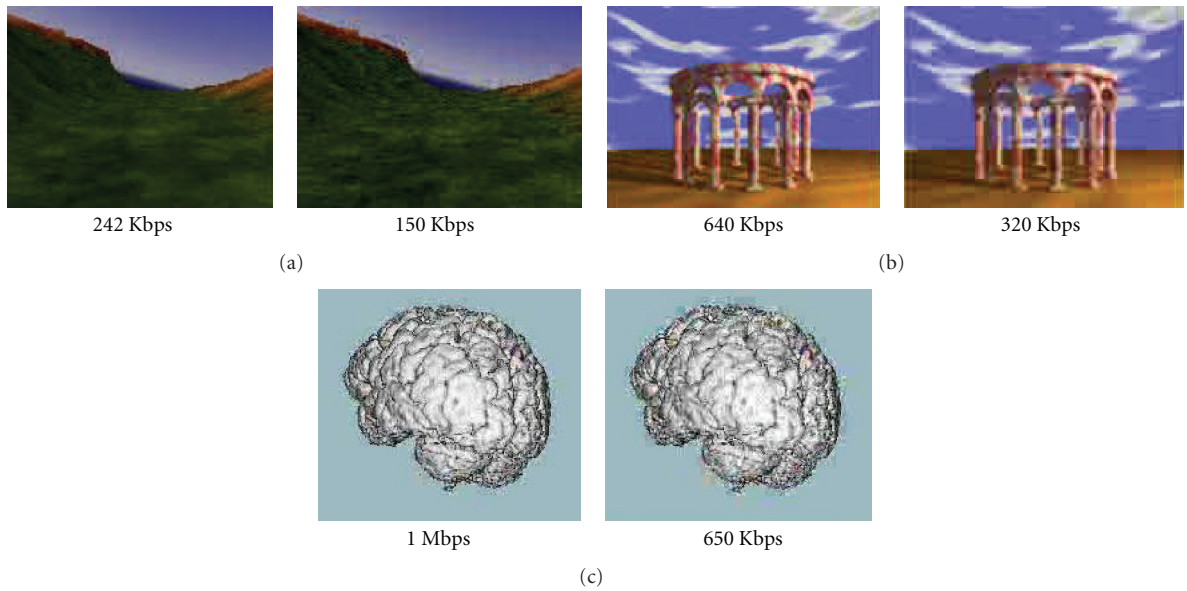


Figure 18: Video quality variation in original and transcoded image (a) canyon.mpeg, (b) rotate.mpeg, and (c) vessel.mpeg.

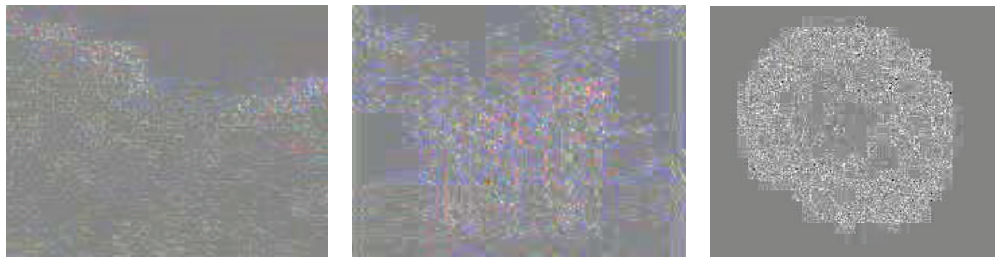


Figure 19: Error images for selected video streams.

In computing RMSE and RE, the RGB values for each pixel in a frame were considered.

Typical PSNR values range from 20 to 40. Transcoded images with higher PSNR and lower PE values are judged to be better. Table 6 summarizes the average PSNRs and PEs for the I, P, and B frames, respectively, of several requantized test video sequences given a 30% reduction in output data rate. The values obtained for each frame type indicate the degree of quantization that was achieved by the transcoder in reducing the bit rate (lower PSNR and higher PE values specify coarser quantization and vice versa).

It is important to note that these error metrics do not always correlate well with perceived image quality though they do provide a good measure of relative quality. A higher PSNR does not necessarily always imply a reconstructed image of better quality. For this reason, actual transcoded images from several streams are provided below to allow the visual effects of quantization to be seen. As the requantization achieves rate reduction by making the DCT values smaller (and some are rounded to zero), the transcoded video loses sharpness. Figure 18 shows the quality variation in several images comparing the original and corresponding transcoded streams.

Another useful technique for demonstrating errors is to construct an error image that shows pixel by pixel errors. Error images are created by taking the difference between the transcoded and original pixels. These error images are difficult to visualize as the difference values are often small and some are zeros which commonly represent black. To make the errors more visible, the difference values are multiplied by a constant and the entire image is translated to a gray level by adding an offset. Figure 19 shows error images for three selected test sequences.

The quality of the transcoded video stream using frame dropping was evaluated by quantifying the smoothness of the motion in the decoded sequence. If the frame (picture) rate is less than 12 fps it is easily detectable by the human eye. As we maintain the minimum frame rate (24 fps) required by the MPEG-1 standard in the frame dropping technique, the transcoded streams exhibit sufficient smoothness of motion in the decoded sequences.

## 11. Conclusions

In this paper, we discussed the design and implementation of a network processor-based transcoder for in-network adaptation of video streams to support collaborative applications. Our transcoder uses the IXP1200 network processor to transcode MPEG-1 streams.

Transcoding was done by requantizing the DCT coefficients with larger quantization values to reduce the bit rate. To verify the practicality of the video adaptation scheme, we conducted experiments and presented results in terms of transcoding latency, throughput, and accuracy of video adaptation. Our video transcoding node implementation can dynamically adapt the test video streams on a packet-by-packet basis at an acceptable rate for use near the network edge (where only a small number of concurrent streams are likely).

Transcoding latencies obtained for test video sequences confirm the real-time performance of the video transcoding. As expected, the speed depends on the bit rate of the compressed video stream; a high bit rate stream incurs higher transcoding latency. We also evaluated the quality of the transcoded video by measuring the average PSNR and percentage error (PE) and through the use of error images.

The current implementation of the transcoder uses the microengines provided by the IXP1200. Involving the StrongARM processor is worth considering for future work as is implementation using more powerful network processors. Our results were not compared to any other system as currently no similar adaptation scheme exists. We did compare the performance of our prototype to an earlier system exploiting only the basic capabilities of the IXP1200 (one thread in each video adaptation microengine) as a baseline.

The prototype adaptation node could be re-implemented relatively easily on more recent IXP series network processors. The use of the IXP2400 or, particularly, the IXP2800 would result in reduced transcoding latency and higher throughput given the increased number and speed of the microengines and due to the larger number of threads supported per microengine. Coding of the adaptation routines would also be simplified on such machines owing to their larger instruction memory capacities. This would improve readability and portability of the code and, possibly, efficiency as well. More aggressive transcoding might also be attempted. Further, the increased capabilities of these newer network processors would be far more effective in supporting the concurrent transcoding of multiple streams (for different multicast transmissions) and/or higher resolution streams corresponding to new encoding standards (e.g., MPEG-2 and H.263). Finally, the significant control processor capabilities of network processors such as the PowerNP and C-5e can be effectively exploited in practical deployments.

## Acknowledgment

This work was supported, in part, by a University of Manitoba Graduate Fellowship.

## References

[1] L. Chen and G. Singh, "Enhancing multicast communication to support protocol design," in *Proceedings of the 11th International Conference on Computer Communications and Networks (ICCCN '02)*, pp. 328–333, Miami, Fla, USA, October 2002.

[2] T. Turletti, "INRIA Videoconferencing system (IVS)," *ConneXions-The Interoperability Report Journal*, vol. 8, no. 10, pp. 20–24, 1994.

[3] R. Frederick, "Experiences with real-time software video compression," in *Proceedings of the 6th International Workshop on Packet Video*, pp. F1.1–F1.4, September 1994.

[4] S. McCanne and V. Jacobson, "Visual audio tool," Lawrence Berkeley Laboratory, ftp://ftp.ee.lbl.gov/conferencing/vat.

[5] S. McCanne and V. Jacobson, "vic: a flexible framework for packet video," in *Proceedings of the 3rd ACM International Conference on Multimedia*, pp. 511–522, San Franscisco, Calif, USA, November 1995.

[6] H. Eriksson, "MBONE: the multicast backbone," *Communications of the ACM*, vol. 37, no. 8, pp. 54–60, 1994.

[7] D. L. Tennenhouse and D. J. Wetherall, "Towards an active network architecture," *Computer Communication Review*, vol. 26, no. 2, pp. 5–17, 1996.

[8] K. L. Calvert, "Architectural framework for active networks," RFC Draft, version 1.0, July 1999.

[9] P. S. Goncalves, J. F. Rezende, O. M. Duarte, and G. Pujolle, "Improving layered video multicast using active networks," Tech. Rep., Pierre and Marie Curie University, Paris, France, March 2001.

[10] R. Ramanujan and K. Thurber, "An active network based design of a QoS adaptive video multicast service," in *Proceedings of the World Conference on Systems, Cybernetics and Informatics*, pp. 643–650, July 1998.

[11] H.-F. Hsiao and J.-N. Hwang, "Layered FGS video over active network with selective drop and adaptive rate control," in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '03)*, vol. 5, pp. 752–755, Hong Kong, April 2003.

[12] L. Cheng and M. R. Ito, "Layered multicast with TCP-friendly congestion control using active networks," in *Proceedings of the 10th International Conference on Telecommunications*, pp. 806–811, March 2003.

[13] H. Akamine, N. Wakamiya, M. Murata, and H. Miyahara, "On the construction of heterogeneous multicast distribution trees using filtering in an active network," in *Proceedings of the 1st International Workshop on Quality of Future Internet Services (QoFIS '00)*, pp. 272–284, Berlin, Germany, September 2000.

[14] R. Keller, S. Choi, M. Dasen, D. Decasper, G. Fankhauser, and B. Plattner, "An active router architecture for multicast video distribution," in *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '00)*, vol. 3, pp. 1137–1146, Tel-Aviv, Israel, March 2000.

[15] R. Wittmann, K. Krasnodembski, and M. Zitterbart, "Heterogeneous multicasting based on RSVP and QoS filters," in *Broadband European Networks and Multimedia Services*, vol. 3408 of *Proceedings of SPIE*, pp. 357–365, Zurich, Switzerland, May 1998.

[16] T. Yamada, N. Wakamiya, M. Murata, and H. Miyahara, "Implementation and evaluation of video-quality adjustment for heterogeneous video multicast," in *Proceedings of the 8th Asia-Pacific Conference on Communications (APCC '02)*, pp. 454–457, September 2002.

[17] Y. Jia, I. Nikolaidis, and P. Gburzynski, "Buffer space tradeoffs in multihop networks," in *Proceedings of the Conference on Communication Networks and Services Research (CNSR '03)*, pp. 74–79, May 2003.

[18] J. L. Mitchell, W. B. Pennebaker, C. E. Fogg, and D. J. LeGall, *MPEG Video Compression Standard*, Chapman & Hall, Boca Raton, Fla, USA, 1996.

[19] "Information Technology—Generic Coding of Moving Pictures and Associated Audio Information—Part 2 (Video)," ISO/IEC 13818-2, 1998.

[20] "Visual: a compression codec for visual data—part 2," ISO/IEC 14496-2, 1998.

[21] H.261, "Video codec for audiovisual services at px64 kbits," International Telecommunications Union Telecommunications Standardisation Sector, ITU-T Recommendation H.261, 1993.

[22] D. H.263, "Video coding for low bitrate communication," International Telecommunications Union Telecommunications Standardisation Sector, ITU-T Recommendation H.263, 1996.

[23] T. Wiegand, G. J. Sullivan, G. Bjøntegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560–576, 2003.

[24] "Information Technology—Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5 Mbit/s—Part 2 (Video)," ISO/IEC 11172-2, 1993.

[25] G. Keeman, R. Hellinghuizen, F. Hoeksema, and G. Heideman, "Transcoding of MPEG-2 bitstreams," *Signal Processing: Image Communication*, vol. 8, pp. 481–500, 1996.

[26] J. Hwang, T. Wu, and C. Lin, "Dynamic frame-skipping in video transcoding," in *Proceedings of the 2nd IEEE Workshop on Multimedia Signal Processing (MMSP '98)*, pp. 616–621, Redondo Beach, Calif, USA, December 1998.

[27] H. Shu and L.-P. Chau, "An efficient arbitrary downsizing algorithm for video transcoding," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 14, no. 6, pp. 887–891, 2004.

[28] Y. Liang, L.-P. Chau, and Y.-P. Tan, "Arbitrary downsizing video transcoding using fast motion vector reestimatlon," *IEEE Signal Processing Letters*, vol. 9, no. 11, pp. 352–355, 2002.

[29] R. Dugad and N. Ahuja, "A fast scheme for image size change in the compressed domain," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 11, no. 4, pp. 461–474, 2001.

[30] "Intel IXP2400 Network Processor—Product Brief," Intel Corp., 2003.

[31] "Intel IXP2805 Network Processor—Product Brief," Intel Corp., 2005.

[32] J. R. Allen Jr., B. M. Bass, C. Basso, et al., "IBM PowerNP network processor: hardware, software, and applications," *IBM Journal of Research and Development*, vol. 47, no. 2-3, pp. 177–193, 2003.

[33] "NP1c Network Processor—Product Brief," EZchip Technology, 2002.

[34] Agere Systems Proprietary, "The challenge for next generation network processors," white paper, April 2001.

[35] "C-5e Network Processor—Product Brief," Motorola, Plantation, Fla, USA, 2003.

[36] M. Shorfuzzaman, R. Eskicioglu, and P. Graham, "Video transcoding using network processors to support dynamically adaptive video multicast," in *Proceedings of the 20th IEEE International Conference on Advanced Information Networking and Applications (AINA '06)*, vol. 1, pp. 471–476, Vienna, Austria, April 2006.

[37] D. S. Alexander, B. Braden, C. A. Gunter, et al., "Active network encapsulation protocol (ANEP)," RFC Draft, July 1997.