MOBILVIDEO: A Framework for Self-Manipulating Video Streams

Ananth Grama, Wojciech Szpankowski, and Vernon Rego

Department of Computer Sciences,
Purdue University, W. Lafayette, IN 47907

{ayg, spa, rego}@cs.purdue.edu *

ABSTRACT

As handheld computing devices and cell phones become commonplace, streaming media to these devices becomes a challenging problem. This is a result of severe bandwidth, processing, and memory constraints imposed by these devices. Bandwidth limitations necessitate effective compression strategies while memory and processing constraints require inexpensive decompression techniques. This need for highly asymmetric compression and decompression techniques is not well addressed by conventional standards such as MPEG. In the absence of a universally accepted standard, it is highly desirable to bundle lean media handlers with the media itself. Advances in infrastructure for mobile code (fast virtual machines, embedded device support) have enabled development of such active streams - media streams capable of self manipulation. In this paper, we describe a highly asymmetric compression/ decompression technique, called 2D-PMC and an associated mobile decompressor called MOBILVIDEO. We demonstrate that the overhead of using mobile code in our framework is minimal, the compression comparable to MPEG2, and decompression possible in real time at 320×240 resolution (the display resolution of an iPAQ 3870) and 20 FPS even on 206 MHz StrongARM class processors.

1. INTRODUCTION

Dramatic growth in networking technology has fueled new research, standards, and products for multimedia compression and decompression. As networking technologies evolve, new challenges are being posed for compression technologies. Thin clients relying on wireless networking emphasize asymmetric compression schemes that are amenable to extremely lean decompression while yielding good compression ratios. Streaming media broadcasts over potentially contested networks require novel and powerful multimedia compression schemes (e.g., [1, 4, 7, 8, 3, 13]). In [1] we proposed novel multimedia compression schemes based on approximate pattern matching that are lossy extensions of the well-known Lempel-Ziv schemes. The central theme of a lossy extension of Lempel-Ziv algorithm is the notion of approximate repetitiveness. If a portion of data recurs in an approximate sense, then subsequent occurrences can be stored as direct or indirect references to the first occurrence.

A highly desirable feature of our scheme is that the associated decompressor is extremely fast and simple. This enables implementation in mobile code that can be shipped along with the media with little communication overhead. Furthermore, even on relatively outdated machines (Pentium, 350MHz), our mobile code achieves real time performance at 25 frames/second on frames of size 360×288 with high motion. We report here, the theoretical basis for our 2D-PMC compression scheme along with experimental results from the MOBILVIDEO decoder. Our experimental results indicate that the 2D-PMC decoder is an order of magnitude faster than an MPEG2 decoder and similarly faster than other streaming video decoders such as VIVOACTIVE, REALSYSTEM G2, and JSTREAMING in native form (machine code). This enables implementation of our decompressor in mobile code (bytecode) while achieving real-time performance.

The implementation of both the 2D-PMC compressor and MOBILVIDEO decoder are challenging problems from the algorithmic and programming standpoints. Finding an efficient data structure for *approximate* search of multidimensional sets in a huge multidimensional database, is an interesting problem in itself. We use a set of modified k-d trees enhanced by generalized run length coding for approximate search. A key issue for high quality image and video

*This work was supported by NSF Grants CCR-9804760 and CCR-0208709, EIA-9806741, ACI-9875899, and ACI-9872101, and contracts from sponsors of CERIAS at Purdue. Computing equipment used for this work is supported by the Intel Corp.

compression is the design of an adaptive distortion measure that automatically adjusts its maximum distortion to produce perceptually high quality results. These findings were partially reported in [1]. For the MOBILVIDEO decoder, we have developed a highly optimized multithreaded implementation that is optimized for small footprint as well as real-time performance.

2. THEORETICAL UNDERPINNINGS

We review and extend here some theoretical results of [1]. Consider a source sequence $(X_{\mathbf{k}})$ taking values from a finite alphabet $\mathcal{A}$ (e.g., $|\mathcal{A}| = 256$), where $\mathbf{k} = (k_1, k_2, \ldots, k_d)$ is a d-dimensional index. That is, $X_{\mathbf{k}} : \mathcal{S} \to \mathcal{A}$ where $\mathcal{S}$ is a two-dimensional area (e.g., for images $\mathcal{S}$ is an $N \times N$ square of pixels). To simplify the presentation, we write X_m^n to denote contiguous block of $n-m+1$ elements (e.g., $X_m^n = X_{k=1,l=1}^{r,s}$ such that $n - m + 1 = r \times s$). We formally should work with *random fields* (e.g., Markov random fields), however, we leave this precise formulation of the problem to an extended version of the paper (the interested reader is refereed to [8] for precise formulation in terms of random fields). Furthermore, we let $P(x_1^n)$ denote the probability of $X_1^n = x_1^n$. We encode the source sequence X_1^n into a compression code and the decoder produces an estimate $\hat{X}_1^n$ of X_1^n. More precisely, a code C_n is a mapping $C_n : \mathcal{A}^n \to \{0,1\}^*$, and we write $C_n(x_1^n)$ for the compression code of x_1^n, where lower-case letters represent realizations of a stochastic process. Let $\ell(C_n(x_1^n))$ be the length of a code (in bits) representing x_1^n. Then, the *bit rate* is defined as $r(x_1^n) = \ell(C_n(x_1^n))/n$ and the *average* bit rate is defined as

$$\mathbf{E}[r(X_1^n)] = \mathbf{E}[\ell(C_n(X_1^n))]/n.$$

In passing we recall that the dimension d is "buried" under our notation since n denotes the total volume in a d-dimensional space; for example, if X^n would denote a sub-block of a d-dimensional cube, then the above expression would have been $\mathbf{E}[r(X^n)] = \mathbf{E}[\ell(C_n(X^n))]/n^d$. We only consider *single-letter fidelity* distortion measures $d : \mathcal{A} \times \hat{\mathcal{A}} \to \mathbb{R}_+$ such that

$$d(x_1^n, \hat{x}_1^n) = \frac{1}{n} \sum_{i=1}^{n} d(x_i, \hat{x}_i).$$

Our implementation of pattern matching video compression is well modeled by the *fixed database model* [12]. In this model, the decoder and the encoder both have access to the common database sequence $\hat{X}_1^n$ (e.g., the first image in the video stream that could be viewed as a three-dimensional data) generated according to the distribution $\hat{P}$. The source sequence X_1^M (e.g., $M = N^2$ for $N \times N$ images) is partitioned according to Π_n into variable length phrases (i.e., rectangles) $Z^1, Z^2, \ldots, Z^{|\Pi_n|}$ of volumes $L_n^1, \ldots, L_n^{|\Pi_n|}$, respectively. More precisely, for given fixed compression bound $D > 0$ we define

$$\begin{aligned} L_n^1 &= \max\{k : d(X_1^k, \hat{X}_i^{i+k-1}) \le D, \ 1 \le i \le n-k+1\}, \\ Z^1 &= X_1^{L_n^1} \end{aligned}$$

This implies that the first partition comprises of the largest d dimensional rectangle of the source X_1^M that matches a rectangle in the database $\hat{X}_1^n$ to the specified tolerance. For example, for 2D data, the string $\hat{Z}^1$ recovered by the decoder is therefore given by:

$$\hat{Z}^1 = \hat{X}_{i_1,i_2}^{i_1+k_1-1, i_2+k_2-1},$$

where $k_1 \times k_2 = L_n^1$. In a similar fashion (cf. [1]) we define a sequence of subsequent partitions Π_n of volumes L_n^m such that the source data X_1^M is parsed as $X_1^M = Z^1 Z^2 \ldots Z^{|\Pi_n|}$, while the decoder recovers $\hat{Z}^1 \hat{Z}^2 \ldots \hat{Z}^{|\Pi_n|}$, which is within distortion D from X_1^M. We represent each $\hat{Z}^i$ by a pointer `ptr` to the database and its lengths. Therefore, its description costs $\log n + O(d \log(L_n^i))$ bits. The total compressed code length is

$$\ell_n(X_1^M) = \sum_{i=1}^{|\Pi_n|} \log n + \Theta(d \log L_n^i),$$

and the bit rate (e.g., in bits per pixel) is given by

$$r_n(X_1^M) = \frac{1}{M} \sum_{i=1}^{|\Pi_n|} \log n + \Theta(d \log L_n^i). \tag{1}$$

To formulate our main result, we introduce the generalized Shannon entropy $\hat{r}_0(D)$ defined as:

$$\hat{r}_0(D) = \lim_{n \to \infty} \frac{\mathbf{E}_P[-\log \hat{P}(B_D(X_1^n))]}{n}, \tag{2}$$

where $B_D(x_1^n) = \{y_1^n : d(y_1^n, x_1^n) \le D\}$ is a d-dimensional ball of radius D with center x_1^n, and $\mathbf{E}_P$ denotes the expectation with respect to P. Our main theoretical result is as follows:

THEOREM 1. *Let us consider the fixed database video model using our 2D-PMC scheme with the database $\hat{X}_1^n$ generated by a Markovian source $\hat{P}$, and the source X_1^M emitted by an independent memoryless source P. Then the average bit rate attains, asymptotically, the following bounds*

$$\hat{r}_0(D) \le \lim_{n \to \infty} \lim_{M \to \infty} \mathbf{E}_{P \times \hat{P}}[r_n(X_1^M)] \le 2\hat{r}_0(D). \tag{3}$$

Proof. Detailed proofs can be found in an extended version of the paper [1] focusing on 2D-PMC compression.

Remark. Our experiments indicate that the l.h.s. is indeed an accurate estimate of the bit rate. ∎

We should point out that $\hat{r}_0(D) \ge R(D)$ where $R(D)$ is the optimal rate distortion function. We observe, at least for memoryless sources, that $\hat{r}_0(D)$ and $R(D)$ do not differ by much for moderate values of D. For an optimal pattern matching compression the reader is referred to [7]. Armed with this theoretical understanding, we have developed a pattern matching compression scheme that relies on approximate matching to yield excellent compression ratios.

3. REVIEW OF PMC VIDEO SCHEME

In this section we review algorithmic and implementation issues of the 2D-PMC video compression scheme. 2D-PMC video compression relies on a range of techniques centered around 2-D pattern matching used in conjunction with variable adaptive distortion and enhanced run-length encoding.

Two dimensional pattern matching is the most efficient and computationally expensive way of compressing images (frames) among the methods used in 2D-PMC. The basic idea is to find a two-dimensional region (rectangle) in the uncompressed part of the image (e.g., the first frame in a group of pictures) that occurs approximately in the compressed part (i.e., database), and to store a pointer to it along with the width and the length of the repeated rectangle. Since the objective is to search for the *largest* such area, a brute force search algorithm is too time consuming. Consequently, we use k-d trees for accelerating search.

Run-length encoding (RLE) of images identifies regions of the image with constant pixel values. We enhance RLE by giving it the capability of coding regions in which pixel values can be (approximately) modeled by a planar function. We call this technique *enhanced run-length encoding* (ERLE). ERLE approximates in the least-squared error sense a given grid $m \times n$ of pixels by a planar surface. The coefficients of the planar surface are computed by solving a system of normal equations associated with the least-squared error procedure. Once the planar surface is determined, a sub-segment of the $m \times n$ grid that is within the distortion distance is identified and coded using ERLE. We observe that this is particularly useful for synthetic images typically found on the Web.

For lossless encoding we use a custom-designed arithmetic encoder. However, to simplify and speed up the decompressor, the arithmetic coder is disabled in the MOBILVIDEO decoder. It is our expectation that as decoders' computational capabilities improve, we can enable arithmetic coding in MOBILVIDEO without losing real-time performance.

These three coding techniques in 2D-PMC (pattern matching, enhanced RLE, and lossless coding) are applied to progressively code images. Assuming that a part of an image (frame) has been previously encoded, the key task is to encode the pixels located to the right of and below a point we call the *anchor point*. The selection of the next anchor point is based on a *growing heuristic* (cf. [3]). The growing heuristic we adopt is the "wave–front" scheme. This scheme sweeps the image from top to bottom and left to right. Other heuristics grow regions in a circular manner from a center of the image or expand from the main diagonal. These heuristics have been observed to yield similar results [3]. Once the anchor point has been identified, the partial image is coded using either 2-D pattern matching, ERLE, or lossless coding. We observe that for all our growing heuristics individually coded subimages do not overlap more than twice.

Our video compression scheme uses a (decompressed) representation of the previous frame that has been compressed in our framework as the database to compress current frame. We refer to this decompressed representation as a lossy image. The reason for using the lossy image as a database is that we want the decompressed image at the client side to be within a constant distortion bound from the original image. Since the decompressor only knows the compressed frame (in its compressed and uncompressed form), this must be used as the database for pattern matching compression. In contrast, if the original previous frame was used as the database, this database is not available at the decompressor. Consequently, errors propagate quickly through subsequent frames during decompression. Since the previous compressed frame (in its uncompressed form) forms a static database, the compression process is modeled well by our analytical framework of Section 2.

4. REAL-TIME DECOMPRESSION USING MOBILE CODE - MOBILVIDEO

A key advantage of 2D-PMC is its highly asymmetric nature. While the computational cost associated with compression is higher due to exhaustive search, the associated decompression cost is much lower since there are no floating point operations. In typical video samples, we observe that compression time is an order of magnitude higher and decompression time is an order of magnitude lower. This low decompression time enables implementation of the decoder in mobile code while achieving real-time performance. This is in contrast to conventional schemes like MPEG that rely on frequency domain transforms defined over fixed size blocks. The associated decompression schemes tend to be expensive and their implementation on conventional java virtual machines (JVMs) renders real-time performance difficult for meaningful frame sizes.

A 2D-PMC compressed file is stored as a script. The three instructions corresponding to pattern matching, enhanced run-length encoding, and lossless coding are stored as two bit instructions followed by their arguments. These are eventually coded using arithmetic coding. The *entire* decompressor method is shown in the listing below. It is evident that the method is extremely lean. In addition to the `decompressPMC` method, the only other significant methods are `readHeader` and `readBuffer`, which read the header of the stream and stream segments in chunks of programmable size (we typically use 2KB segments). A detailed demonstration of the mobile decompressor is available at `http://www.cs.purdue.edu/homes/ayg/Video/Videos` and the decompressor code itself is available at `http://www.cs.purdue.edu/homes/ayg/Video/Videos/Player-0.99/`.

```
public void decompressPMC() {
  Date        dStart  = new Date();
```

```
Date        dEnd;
int       iMin, iSec;
int loop;

while (numOps != 0) {
  dStart = new Date();
  if (readIn == 0 &&
      byteOffset >= LEFT_BUFFER) {
    readBuffer(0, LEFT_BUFFER);
    readIn  = 1;
  }

  switch (getByteSeg(SIZE_OF_OP)) {
    case 0 :
       writePixel_BW(getByteSeg(SIZE_OF_LX));
       break;
    case 1 :
       writeRLE_BW(getByteSeg(SIZE_OF_LX),
              getByteSeg(SIZE_OF_LY),
              getByteSeg(SIZE_OF_C0)-256,
              getByteSeg(SIZE_OF_CX)-256,
              getByteSeg(SIZE_OF_CY)-256);
       break;
    case 2 :
       writePattern_BW(getByteSeg(SIZE_OF_LX),
              getByteSeg(SIZE_OF_LY),
              getByteSeg(SIZE_OF_WID)-512,
              getByteSeg(SIZE_OF_HI) -512);
       break;
  }
  numOps--;
  if (pos >= MAX_POS) {
    drawImage();
    dEnd = new Date();
    if ((dEnd.getTime() - dStart.getTime())
                                      < 40) {
      loop  = 4000;
      while(loop > 0) { loop--; }
    }
  }
}

dEnd = new Date();
iMin = dEnd.getMinutes()-dStart.getMinutes();
iSec = dEnd.getSeconds()-dStart.getSeconds();
if (iMin < 0)
  iMin += 59;
if (iSec < 0) {
  iMin++;
  iSec += 59;
}
  graph.setColor(Color.white);
  graph.drawString("mm:ss " + iMin + ":"
                    + iSec, 0, 120);
  iSec = (int)(dEnd.getTime() -
                           dStart.getTime());
  graph.drawString("Milli " + iSec, 0, 140);
}
```

Figure 1: Screenshot of the MOBILVIDEO **decompressor illustrating video without plugins with excellent compression and decompressor performance.**

The MOBILVIDEO decompressor for 2D-PMC is fashioned as a Java applet that communicates the image or video segment, decompresses it, and renders it. To optimize the performance of MOBILVIDEO, its functionality is implemented in three threads - `reader`, `decompressor`, and `render`. These threads communicate via synchronized (mutually exclusive) operations on shared buffers. The reader and decompressor threads communicate via the input (in) buffer and decompressor and render threads communicate via the display buffer. Compressed data is copied into the input buffer in (programmable) chunks of size 2KB. The size of this chunk is selected to optimize network latency, buffer management overheads, and compressed image sizes. Decompressed data is copied into the display buffer one frame at a time. A screenshot of the MOBILVIDEO decompressor is illustrated in Figure 1.

The reader thread reads a parcel of data and places it in the input buffer if space is available, otherwise it yields (condition waits). The decompressor reads in data from the input buffer as it becomes available. If there is no data in the input buffer, it yields as well. If there is data, it attempts to decompress the data and place it into the display buffer when an entire frame has been decompressed. If the display buffer is full, the decompress thread yields as well. The render thread reads in frames from the display buffer as they become

available and renders them at programmed intervals. Threads are programmed to maximize concurrency and minimize synchronization overheads. Buffer sizes are selected to optimize serialization overheads as well as for latency tolerance.

5. EXPERIMENTAL RESULTS

In this section, we present detailed experimental results corresponding to compression ratios, compression time, and decompression time. The objective of this exercise is to demonstrate that 2D-PMC achieves compression rates similar to MPEG2 while supporting decompression rates roughly an order of magnitude higher. The corresponding compression rates are roughly an order of magnitude lower for 2D-PMC. We demonstrate these in the context of a variety of video segments.

Our experimental demonstrations are based on five video segments, each with distinct characteristics. Samples Claire and Missa correspond to news broadcasts. The background is static and the motion is limited. Consequently, most compression techniques yield excellent compression for these video segments. Segments Football, PomPom, and Ping-Pong (Figure 2) have significant motion in them and some scene changes. Segment Train (Figure 3) corresponds to a cartoon clip. This segment is selected to demonstrate the superior characteristics of 2D-PMC for synthetic video segments.

5.1 Compression Rates for MPEG2 and 2D-PMC

Our first experiment compared the compression rates achieved from MPEG2 and 2D-PMC. We selected a group of pictures to correspond to ten frames in each case. Data rates were computed from the sizes of each group of pictures. The parameters in 2D-PMC were selected to ensure that video quality is comparable to or better than that of MPEG2. For a full demonstration, we direct the reader to `http://www.cs.purdue.edu/homes/ayg/Video/`. We observe from Table 1 that the performance of MPEG2 and 2D-PMC is comparable for a wide range of segments with variation from 7.9% worse (in case of Claire) to 31.3% better (in case of synthetic video segment Train).

2D-PMC has a number of programmable parameters such as search region, number of seed points, number of templates, etc., that allow tradeoffs between compression time, video quality, and compression ratios. The results presented in Table 1 are selected as the best compromise between image quality and compression time. If a better compression ratio is desired, it is easy to change the parameters of the compression script to explore better matches at the expense of increased compression time.

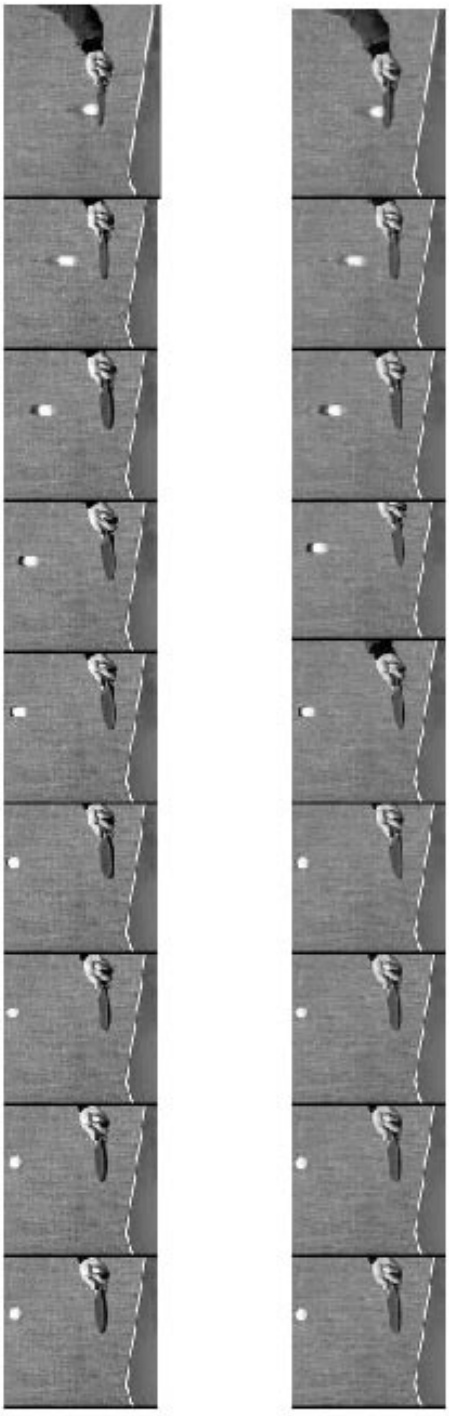

Figure 2: Sample video Ping Pong – original (left) and PMC compressed (right) for illustration.

Figure 3: Sample video Train – original (left) and PMC compressed (right) for illustration.

Sample	MPG	PMC	Comp. Time		Decomp. Time	
			MPG	PMC	MPG	PMC
Claire	17.7	19.1	2	26	0.36	0.05
Football	111.9	90.9	3	29	0.34	0.09
Missa	20.4	20.2	9	23	0.32	0.03
PomPom	187.1	174.6	7	34	0.35	0.07
PingPong	113.8	104.9	8	39	0.35	0.03
Train	202.8	139.3	9	25	0.35	0.04

Table 1: Comparison of data rates (KB/s), compression, and decompresion times (in seconds on a Pentium, 350MHz, 128MB) from five different samples illustrating that 2DPMC yields performance ranging from 7.9% worse to 31.3% better than MPEG2. The sizes of samples Claire, Football, Missa, Pom-Pom, Ping Pong, and Train are 360×288, 360×243, 360×288, 352×288, 360×256, and 352×288 pixels respectively. The platform is intentionally chosen to be a computationally weak one (the weakest we could find in our lab) to demonstrate the highly assymmetric nature of 2D-PMC.

5.2 Compression Times for MPEG2 and 2D-PMC

The compression times for MPEG2 and 2D-PMC are presented in Table 1. The compressors are implemented in native code as opposed to mobile code. This is motivated by the assumption that servers are likely to have significant computational resources. We observe that 2D-PMC is 3- to 10-times slower than MPEG2 in terms of compression time. The best performance for 2D-PMC is for synthetic segments (e.g., Train).

The significant time premium of 2D-PMC can be attributed to its more exhaustive search. This search time is a sensitive function of various parameters - the number of seed templates, search region for seed points, and number of seed points. In general, significant improvements in compression times can be achieved with minimal degradation in compression ratios.

5.3 Decompression Rates for MPEG2 and 2D-PMC

Decompression times represent the most significant advantage of 2D-PMC over existing compression standards. Since decompression in 2D-PMC is simply a set of pointer lookups or increment operations in case of ERLE, it is extremely fast. In Table 1, we compare the decompression times for MPEG2 and 2D-PMC. The times correspond to total decompression time for a group of ten pictures using native (machine) code instead of mobile code. This is because a mobile decompressor for MPEG2 was unavailable for comparison, therefore, for a fair comparison, native code was used in both cases. It is evident from the table that MPEG2 decompression takes about 34ms/frame on an average whereas 2D-PMC takes

roughly 4 ms/frame. This implies that our mobile implementation of a 2D-PMC decompressor easily achieves real-time performance even on older platforms such as a Pentium 266/Windows platform.

5.4 Discussion of Performance and Related Systems

There have been other efforts at developing mobile decompressors for a variety of compression techniques. The closest in terms of project objectives is the JSTREAMING H263 decoder from the Multimedia Communications Research Lab at the University of Ottawa (`http://www2.mcrlab.uottawa.ca/jauvane/H263Decoder/JDK1.1/`). Being a H263 decoder, it is suited for low bit-rate low-motion video. Similar products are also available from EMBLAZE (`http://www.emblaze.com`). There have also been efforts at mobile MPEG2 decoders. However, the performance of these decoders is far from real-time for higher quality video.

6. REFERENCES

[1] M. Alzina, W. Szpankowski and A. Grama, 2D-Pattern Matching Image and Video Compression, *IEEE Trans. on Image Processing*, 11, 318-331, 2002.

[2] G. Conklin, G. Greenbaum, K. Lilevold, A. Lippman, and Y. Reznik, Video Coding for Streaming Media Delivery on the Internet, *IEEE Transactions on Circuits and Systems for Video Technology*, 2001, to appear.

[3] C. Constantinescu and J. A. Storer, Improved Techniques for Single-Pass Adaptive Vector Quantization, *Proc. IEEE*, 82, 933-939, 1994

[4] W. Finamore, M. Carvalho, and J. Kieffer, Lossy Compression with the Lempel-Ziv Algorithm, *11th Brasilian Telecommunication Conference*, 141-146, 1993.

[5] J. Gibson, T. Berger, T. Lookabaugh, R. Baker *Multimedia Compression: Applications & Standards*, Morgan Kaufmann Publishers 1998.

[6] JSTREAMING - H263 Video Decoder 1.8.

[7] I. Kontoyiannis, An Implementable Lossy Version of the Lempel-Ziv Algorithm– Part I: Optimality for Memoryless Sources, *IEEE Trans. Information Theory*, 45, 2285-2292, 1999.

[8] I. Kontoyiannis, Pattern Matching and Lossy Data Compression on Random Fields, preprint 2002.

[9] T. Łuczak and W. Szpankowski, A Suboptimal Lossy Data Compression Based in Approximate Pattern Matching, *IEEE Trans. Information Theory*, 43, 1439–1451, 1997.

[10] W. Szpankowski, *Average Case Analysis of Algorithms on Sequences*, John Wiley & Sons, New York, 2001.

[11] Vivo Software, VIVOACTIVE software documentation, `http://www.vivo.com`.

[12] A.J. Wyner, The Redundancy and Distribution of the Phrase Lengths of the Fixed-Database Lempel-Ziv Algorithm, *IEEE Trans. Information Theory*, 43, 1439–1465, 1997.

[13] E.H. Yang, and J. Kieffer, On the Performance of Data Compression Algorithms Based upon String Matching, *IEEE Trans. Information Theory*, 44, 47-65, 1998.

[14] Z. Zhang and V. Wei, An On-Line Universal Lossy Data Compression Algorithm via Continuous Codebook Refinement – Part I: Basic Results, *IEEE Trans. Information Theory*, 42, 803-821, 1996.