International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:1, No:12, 2007

Linux based Embedded Node for Capturing, Compression and Streaming of Digital Audio and Video

F.J. Suárez, J.C. Granda, J. Molleda, and D.F. García

***Abstract*—**A prototype for audio and video capture and compression in real time on a Linux platform has been developed. It is able to visualize both the captured and the compressed video at the same time, as well as the captured and compressed audio with the goal of comparing their quality. As it is based on free code, the final goal is to run it in an embedded system running Linux. Therefore, we would implement a node to capture and compress such multimedia information. Thus, it would be possible to consider the project within a larger one aimed at live broadcast of audio and video using a streaming server which would communicate with our node. Then, we would have a very powerful and flexible system with several practical applications.

***Keywords*—**Audio and video compression, Linux platform, live streaming, real time, visualization of captured and compressed video.

I. Introduction

THE growing processing capacity of modern computers enables a common PC to do tasks that only big mainframes could handle a few years ago. Modern personal computers are multimedia centers which can play a music CD and even view live broadcast video from Internet. This task, the broadcasting of multimedia contents through the Internet, has undergone a great impulse in the last years. Nowadays, we can find several tools for streaming audio and video [1]–[4].

Streaming [5] is divided into two categories depending on the origin of the data to be broadcast: the broadcast of previously processed multimedia data, which is often transmitted on demand, and the live broadcast of audio and / or video. In the first case, multimedia data stored in a server is broadcast through the Internet, i.e. broadcast of films on demand (Pay per View). The latter, live streaming consists of broadcasting data as it is received, normally from an external source. Live broadcast of a music concert through the Internet is a clear example of this technique.

Multimedia data must be compressed in order to broadcast it on demand or live through the Internet. Otherwise, the size of the broadband required would be prohibitive.

In live streaming, the processing capacity necessary to capture and compress audio and video in real time is very high. Moreover, the bandwidth required for transmitting even compressed multimedia data is very significant. The level of compression applied to the information should be high enough to reduce this requirement. As a result, the loss of quality of the final information, as compared to the source, will be high too. Thus, we must apply efficient compression of the multimedia data in two terms: it shall have low broadband requirements and its quality must be acceptable.

Our final goal is to send multimedia data from a capturing and compressing node running Linux to a live streaming server.

In this paper we focus on the development of an application prototype which will allow us to capture and compress audio and video in real time based on open code to run in an embedded system. Also, it shall allow us to play the captured video and compress it in real time to view it at the same time as the original. This will show the differences between the two videos and the quality loss introduced by the compressing process. The audio can be compared in the same way as the video, although two audio sequences cannot be played at the same time. In addition it will give us some information as to how suitable the compression process is to real time. In Fig. 1 we can see the elements forming a video broadcasting system.

There are few tools on the market providing similar functionality. In the Linux architecture we can find MEnconder [20], which allows us to capture and compress audio and video but always from a command line, so we cannot see the final result until the compression or capturing process is finished.

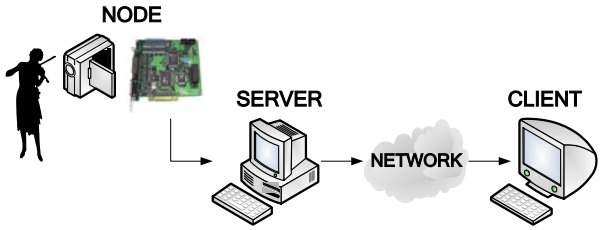

Fig. 1 Live streaming system

In Windows architectures there are a huge number of tools for this kind of purposes. The most representative tool (only open source) is VirtualDub [21]. This tool presents two windows in which we can see both the captured and the

Manuscript received July 14, 2005. This work was supported in part by a Research Program of Spanish Government under Project TIC2001-3595.

Authors are with the Computer Engineering Group, Computer Science Department, University of Oviedo, 33204 Gijón, Spain (phone: +34 985182223; fax: +34 985181986; e-mail: fjsuarez@uniovi.es).

compressed video, but at a slower frame rate than the original, which makes it more difficult to compare the two videos.

The organization of the rest of the paper is detailed below. Section II shows the specifications the prototype must follow. In section III we take into account the multithread architecture of the prototype which is obtained from the specifications it must fulfil. In section IV the multiple technological possibilities to capture audio are considered. Section V describes different options to capture video. In section VI a solution for compressing the captured audio and video is presented. In section VII we analyze the problems introduced by the synchronization of two captured data streams. Section VIII approaches the necessary synchronization for the visualization of two videos at the same time: the original captured video and the compressed one. Finally, in section IX we will extract conclusions from the development of the prototype and its operation. In addition, we will suggest possible future works to build a complete system of streaming of audio and video through the Internet.

II. System Specifications

The prototype tool under development must fulfil a series of requirements:

1) Capture of video: It is able to capture video in real time with different standard formats and with different frame rates.
2) Capture of audio: It is possible to capture audio with various numbers of channels (mono, stereo), numbers of samples per second, etc.
3) Compression of video: It is able to compress the video signal using different *codecs*, especially those following the MPEG-4 standard [7], e.g. [8] and [9], in order to save the data to disk in Audio Video Interleave (avi) format [10]. In the future, it will be possible to broadcast the data to a streaming server instead of saving it to disk.
4) Compression of audio: It is able to compress the captured audio using different audio codecs. We are mainly interested in the use of MPEG-1 Layer 3 codecs [19], e.g. [11].
5) Viewing of video: It views both the original captured video and that resulting from the compression process at the same time, in order to compare quality.
6) The source code of every module used for developing the prototype must be open source. It is possible to optimize it, removing non-essential parts, to make it run in an embedded system.

Due to the real time characteristics desired in the implementation of the prototype, several considerations must be taken into account, i.e. data compression time and its saving-to-disk time, in order to meet goals of functionality and throughput, a multithread prototype has been developed. The essential necessity of task parallelization: the synchronization of different threads and processes, will make the final implementation more difficult.

In Fig. 2 we can see the different computational elements which make up the system. Below, we will see how these elements will become threads in the final prototype.

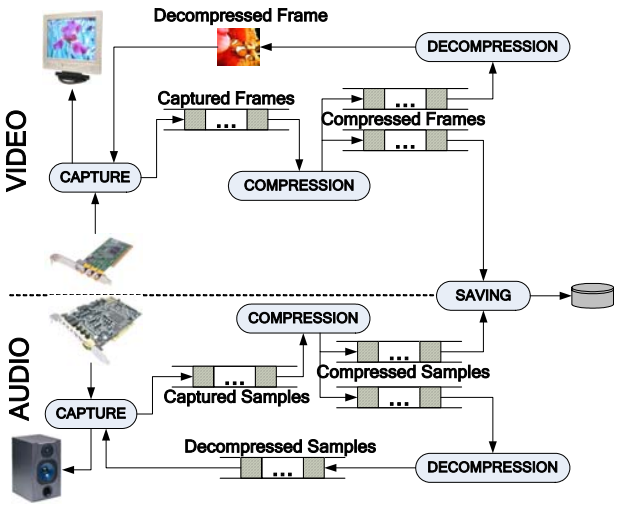

Fig. 2 Computacional elements of the prototype

III. Multithread Architecture

Most of the tasks the prototype will carry out need a great many resources, sometimes even exclusive attention from the process. Thus, functionality must be broken down into small tasks. These tasks will be implemented as threads, so we will have an efficient modular design, whether or not it runs in a multiprocessor.

A server-client relation will be held between different threads. The client is blocked, waiting for any data the server sends through a fifo queue.

The functions of each thread of the prototype are described below:

A. Video Capture Thread

One thread must be dedicated to capturing video. The thread is blocked until a new video frame is ready. Once the frame is received, it is inserted in a fifo queue as long as the compression thread is running to compress it (which may not be necessary depending on the configuration options of the prototype). If a video is not being saved to disk nor a compressed video being viewed, then it is not necessary for the frame to be processed or inserted in the queue.

B. Audio Capture Thread

The mission of the capture thread is the same as the previous thread mission. It must be blocked in a similar way as the video thread, waiting for audio samples in constant time periods. Next, it inserts them in a fifo queue to compress them later. This thread must play the captured or the compressed audio, depending on which audio sequence the user wants to play.

C. Video Compression Thread

This thread compresses video data using a codec selected

by the user. It is blocked in a semaphore until the video capture thread inserts a frame in the queue and sends a signal to the semaphore. Once the thread is unblocked and extracts a video frame from the queue, it compresses the frame using the aforementioned codec and acts as a compressed frame server to the decompression and saving threads.

There must always be tow queues, one for each thread, as both of them need to process all frames. With only one queue, if one thread extracts a frame from the queue, that frame would not be available for the other thread. However, the two queues must be implemented without duplicating memory.

D. Video Decompression Thread

This thread waits for the video compression thread to deliver compressed video frames. Once the compressed video frame is received, this thread decompresses it. Where possible, the same codec used for compression is used for decompression. Otherwise, an alternative one is used.

After the video frame is decompressed, it is passed to the video capture thread for viewing.

E. Saving to Disk Thread

This thread receives compressed video frames and audio data and saves them together at the same time in an avi or wav file, depending on the presence of audio data. It is important to maintain synchronism between audio and video.

The saving thread is a client of the video compression and the audio capture threads. We cannot use the communication method used above to transfer data from two server threads to a client. The thread may be blocked waiting for data in a queue while it is receiving data from the other queue. Thus, a non-blocking method must be used.

This problem can be solved by introducing the concept of audio or video guided capture. This consists of identifying which data stream is used to determine the point of synchronization. This will be discussed in the following sections.

IV. Audio Capture

The audio capture mechanism must be flexible enough to capture audio from different sources with several qualities.

On the Linux platform we can find two standards providing audio support to the operating system: Open Sound System (O.S.S.) [12] and Advanced Linux Sound Architecture (A.L.S.A.) [13]. A.L.S.A. is designed to replace the older Standard, O.S.S., offering backwards compatibility and adding new functionality. Thus, the audio capture process is implemented using ALSA-lib [14], which is a library providing A.L.S.A. native support to user applications.

Another advantage of using A.L.S.A., essential in our case, is that its code is open source, whereas O.S.S. is a commercial implementation. On the other hand, A.L.S.A. is only available on Linux platforms, so the portability of the applications following this standard will be reduced. However, this is not important because we are sure about the platform on which our prototype will be run.

The audio capture and playback process consists of defining a capture or playback buffer whose temporal size can be set by the application. Audio captured or outgoing samples will be stored in this buffer until the application receives them or the audio hardware reads them. Audio is captured in constant time periods.

V. Video Capture

The architecture offers the applications various possibilities for capturing video. Most of drivers used in video capture on Linux platforms follow any of the Video for Linux standard (V4L) versions [17]. Currently, there are two incompatible versions, V4L and V4L2, so we must either provide support for both versions, or decide which one we follow. In this case, we have decided to use V4L2 because all modern Linux systems implement V4L2, since the introduction of kernel 2.6.x. Moreover, V4L2 is designed to replace V4L.

Having established the version of Video for Linux, we must choose a video capture driver to follow it, capable of communicating with the capture card. We use a card with a Brooktree [16] chip which is very popular and widely used in many televisions tuner cards.

The driver chosen is the popular Bttv [17], well known in the Linux World, providing support to any of the cards based on the Brooktree chip.

VI. Audio and Video Compression

In order to compress captured audio and video we must communicate with codecs. The problem is that there is no standardized Application Program Interface (API) to access codecs on Linux platforms. Every codec defines its own API, so the use of several codecs with different APIs increases the difficulty of the final implementation.

Another problem with codecs is that there are few codecs with a Linux version; instead they are usually developed for other platform. Ideally, we could use this kind of codecs, e.g. codecs for Windows platforms.

To solve these two problems we decided to use the AVIFILE [18] library, which allows access to different codecs. AVIFILE is an open source multiplatform library which provides a work frame to manipulate multimedia data using avi files. It can access codecs using plug-ins implementing a common API for any codec. Furthermore, it is possible to set up most of the parameters of these codecs. Another interesting characteristic of AVIFILE is that it allows us to access to Windows codecs distributed in Dynamic Link Library (dll) files.

On the other hand, it is a little documented library so the user must invest time in learning its management. Moreover, some of the plug-ins that comes with it presents undocumented faults which decrease the power of the codecs.

AVIFILE was released in 2000 and nowadays is most commonly used for playing films compressed with DivX in an acceptable way on a Linux platform. The library has been greatly developed and now supports a great number of audio

codecs, as well as video codecs.

It is easy to add new codecs to the library. A version of the codec for the Windows platform in a dll file can be put in a specific directory. On the other hand, if the codec is Linux native, we must develop a plug-in for the library in order to use it.

VII. Synchronization

In order to obtain satisfactory results using two different continuous data streams, they must be synchronized. For example, an undesired effect in audio and video broadcasting occurs when the streams are not synchronized and the audio does not match the facial movements of the speaker.

In our prototype application we can find two different types of synchronization. The first involves saving to disk or broadcasting using streaming of compressed audio and video. The second involves viewing the captured and compressed video frames simultaneously in order to verify the quality decreased through the compression process. The latter will be discussed in following sections.

The first type of synchronization, involving two compressed streams for streaming, can be split into two: the stream level synchronization and the intra-stream synchronization. In the first, all the streams to be broadcast must begin and end at the same time instants. That is, they must begin at the same time and have the same duration. The mechanism of synchronization can be seen in Fig. 3

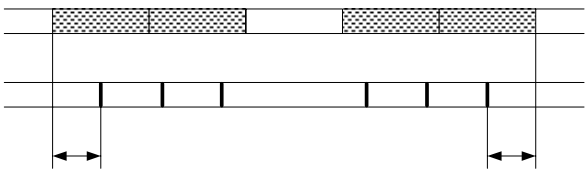

Fig. 3 Two streams synchronization

The synchronization of audio and video captured data in order to save them to disk is carried out by the saving to disk thread. It takes audio and video data from the two queues and it saves them in an interlaced way using timestamps.

In Fig. 3, the video stream can be seen as a discontinuous segment of data whereas the audio stream is continuous. The synchronization process is guided by the video stream as it determines when the capture process starts and finishes. In other words, the audio thread will start before the video and will finish later.

As both of the streams start at different time instants, it is necessary to truncate that which begins first: in our case, the audio stream. This is called the initial synchronization.

In the same way, the final time moment of the streams does not match, so that which ends later must be truncated. This process is called the final synchronization.

The steps necessary for these synchronizations are discussed below.

A. Initial Synchronization

The streams should begin at the same time. We use the video stream to determine when the capture process must start, that is, we take the first video frame out the queue and, depending on its timestamp, we reject all audio data captured before this time. If no audio data is available in the queue, we reject the video frame and we restart this process. Once we find the audio period corresponding to the video frame timestamp, we reject all the samples within the period before the timestamp, so only those captured later are accepted.

B. Final Synchronization

As in initial synchronization, one stream must be truncated, in this case at the end. We save all captured audio periods before the last video frame. The last period must be truncated in order to remove those audio samples later than the time determined by the video frame timestamp.

One might think that it would be better to truncate the video stream instead of the audio stream. However, if we decide to make an audio stream guided capture of a video in which one frame per second were being captured, we might miss the first frame of the video sequence. As a result, we may not capture what the user wanted.

Once the beginning and the end of the streams are determined, it is necessary to interlace data into an avi file, for playback. This kind of synchronization it is called intra-stream.

VIII. Simultaneous View

Another kind of synchronization in our prototype involves simultaneous viewing of the captured and the compressed video. Of course, playing the captured and the compressed audio could be considered a new kind of synchronization, but in fact it is not because we can only play one audio sequence at the same time, discarding the audio samples of the audio sequence not being played.

In order to view the compressed video, we need to compress the captured video and then decompress it to compare it with the original one. This implies an extra overhead in the system which may not be admissible in certain situations, depending on the characteristic of the video (frames per second, size, image movements, compression level, etc.).

Therefore, it is necessary to evaluate the computational time required for the compression and the decompression of the different video frames. The sum of these two times must be less than the exposure time of each video frame, making the processor utilization factor less than 1. In this case, the viewing of the compressed video can take place in real time.

A. Captured Video Viewing

A video sequence is a collection of video frames. The period of time between consecutive video frames is constant and it is the inverse of the frame rate. This is very important when viewing video in a computer: we must keep the original timing between two consecutive video frames.

The multithread architecture of the prototype is fundamental. There is one thread capturing video, another for capturing audio, and a third one for saving data to disk. The video capture thread receives the images which compose the video sequence periodically. This period time might not always be the same, as it may be affected by an error due to the stochastic of the computing systems. Considering this error as negligible (but inevitable), the same thread used for video capturing can also be used for rendering images on the screen. As soon as a new video frame arrives and is added to the fifo queue, it is rendered. So the video is rendered on screen at its nominal frame rate, but with a small error corresponding to the computational time necessary to add the video frame to the queue and render it. However, this time is still negligible.

B. Compressed Video Viewing

In the case of viewing the compressed video, another method is needed. The original captured video frame must be compressed before decompressing it to render it on the screen. The delay introduced by this extra computation may derive in synchronism loss.

Depending on the time necessary to compress and decompress each video frame, two cases can be detailed:

1) The sum of the computation times of the compression and the decompression of the video frame is greater than the exposure time of the video frame. In this case, it is not possible to view the compressed video in real time, as the time necessary to process a frame is greater than the exposure time. The utilization factor of the processor would be greater or equal to 1 so the original frame rate could not be kept.
2) The computation time necessary to compress and decompress the video frame is less than the exposure time of the frame. In this case, the viewing of the compressed video can be carried out in real time. While one video frame is being rendered, another can be processed for rendering.

Hereafter, only the second condition will be accepted. Otherwise, the compressed video cannot be viewed in real time. If this condition were not met, the computer on which the prototype is being run must be upgraded to reach compatible computational capacity.

The times needed to compress and decompress a video frame, are not always constant, but may vary greatly from one frame to another. For example, the compression time of a Key-Frame (the whole image instead of differences between successive frames or movement vectors) needs a much higher compression time than that needed to compress other kinds of frames. Moreover, the time may change depending on the type of the video to be compressed, e.g. frame rate, size, etc.

C. Delay of the Video Stream and Simultaneous Viewing

As we saw in previous sections, delay in viewing the captured video in contrast to the original one is negligible as it is not detectable by the user.

On the other hand, the delay produced viewing the compressed video in contrast to the captured one is significant, so we must approach to the problem of the synchronization of the two videos in two different ways:

1) The delay produced between one and the other is considered negligible and we do not need a higher level of synchronism.
2) The delay is considered to be significant and real synchronization is needed between the two video sequences.

In our case the second supposition is true. One solution is to introduce an extra delay in both videos in contrast to the original one. This is like calculating the lowest common multiple of the delays present in the captured and the compressed videos. This can be done by viewing the previous video frame at the moment that the next frame is being captured. Then, we insert enough time to allow the previous frame to be compressed and decompressed (supposing that the time to carry out these two operations is less than the exposure time).

Therefore, it is necessary to have a buffer to store the previous video frame and view it only when the next is captured. Thus, the fact of capturing a video frame is useful to synchronize the two video sequences. As a result, the video capture thread has to play back both the captured and compressed video sequences while it is capturing the different video frames. In Fig. 4 the most important windows of the prototype can be seen while comparing both the captured and the compressed video streams.

The compressed audio can be played at the same time as the compressed video. The multithread structured designed for the playback of video must be duplicated, so the audio must be decompressed in order to play it. A new thread would be necessary to decompress the audio and feed the audio hardware with audio samples.

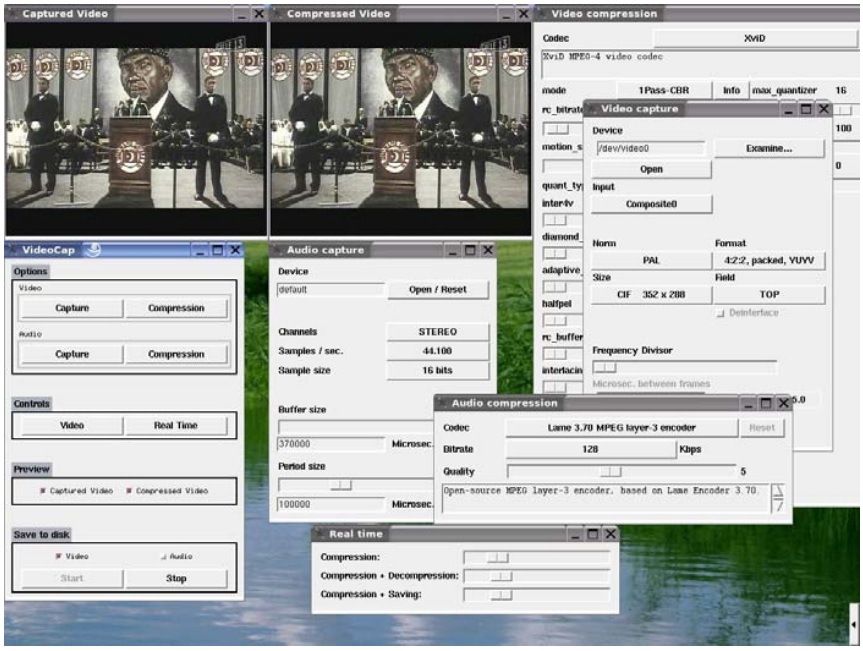

Fig. 4 The most important windows in the prototype

IX. Conclusions

We have developed a prototype application which is designed to implement a node for capturing and compressing audio and video in real time on a Linux platform. This node will communicate with a streaming server in order to provide a live broadcasting service. Moreover, we have developed extra functionality to view the captured and the compressed videos simultaneously in order to we can compare the qualities of both videos.

Several tests have been done, mainly using two video codecs, DivX and XviD, with positive results. The compression and the decompression in real time can take place with a moderate level of occupancy of the processor.

The weak point of the prototype is that it can only manipulate old fashioned codecs (circa 2002-03 versions), because of the long delays between new releases of the AVIFILE library.

Future work may include the possibility of communicating with a streaming server to implement a live broadcasting video server. Moreover, a mechanism to objectively evaluate the quality of the compressed video could be added, e.g. Peak Signal to Noise Radio (PNSR) [6] which measures the power of the resulting signal.

References

[1] Quicktime Streaming Server [Online]. Available: http://www.apple.com/quicktime/products/qtss/
[2] Darwin Streaming Server [Online]. Available: http://developer.apple.com/darwin/projects/streaming/
[3] Icecast Streaming Media Server [Online]. Available: http://www.icecast.org/
[4] Fluid Streaming Server [Online]. Available: http://fluid.sourceforge.net/
[5] D. Wu, Y. T. Hou, W. Zhu, Y. Zhang and J. M. Peha. "Streaming video over the Internet: Approaches and directions". *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 11, no 3, pp. 1-19, March 2001. Available: http://ieeexplore.ieee.org/iel5/76/19666/00911156.pdf
[6] MEncoder. MPlayer [Online]. Available: http://www.mplayerhq.hu/homepage/index.html
[7] VirtualDub [Online]. Available: http://www.virtualdub.org/
[8] Overview of the MPEG-4 Standard [Online]. Available: http://www.chiariglione.org/mpeg/standards/mpeg-4/mpeg-4.htm
[9] XviD codec [Online]. Available: http://www.xvid.org/
[10] Divx [Online]. Available: http://www.divx.com/
[11] AVI File Format Specification [Online]. Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directshow/htm/avifileformat.asp
[12] MPEG-1 Standard [Online]. Available: http://www.chiariglione.org/mpeg/standards/mpeg-1/mpeg-1.htm
[13] L.A.M.E. mp3 encoder [Online]. Available: http://lame.sourceforge.net/
[14] Open Sound System [Online]. Available: http://www.opensound.com/linux.html
[15] Advanced Linux Sound Architecture [Online]. Available: http://www.alsa-project.org/
[16] ALSA-lib [Online]. Available: http://www.alsa-project.org/alsa-doc/alsa-lib/
[17] Video for Linux [Online]. Available: http://linux.bytesex.org/v4l2/
[18] Conexant Systems [Online]. Available: http://www.conexant.com/
[19] Bttv driver [Online]. Available: http://linux.bytesex.org/v4l2/bttv.html
[20] Linux AVI file Library [Online]. Available: http://avifile.sourceforge.net/
[21] Peter M. Kuhn and et al. "Complexity and PSNR-comparison of several fast motion estimation algorithms for MPEG-4". *Technical University of Munich*, 1998.