advantage of CMT's superior flow control and the list of frame-level statistics. For example, we can add a dialog or slide bar that supports random access to any given frame. In addition, we can use the frame sizes and frame rate information provided to immediately identify and bring to the user's attention areas where the actual bitrate exceeds the target bitrate. In the current version of **mpeg_bits** this is not possible since the size of a frame is not known until the frame is parsed and displayed. Using the "getStats" command, this information can be obtained before the stream even begins to play.

As the CMT project continues, these extensions may have to be updated to keep pace. There is currently interest in modifying the file object to parse an MPEG-1 stream to construct the frame index on the fly or on demand as opposed to creating the entire index when the file is opened if the index does not already exist. The problem with the current approach is that for a long stream, for example a feature-length film, constructng the index takes an unacceptably long time. The proposed solution is to construct segments of the index when needed so that parsing is less noticeable. With these modifications, the functionality of the getStats command will have to be modified to behave in a similar fashion, supplying as much feedback as is available and updating the list on the fly as more segments are parsed.

References

[1] K. Gong and L. A. Rowe. "Parallel MPEG-1 Video Encoding," Prod 1994 Picture Coding Symposium, Sacramento, CA, September 1994.

[2] MPEG-1 FAQ version 3.2. ftp://mm-ftp.cs.berkeley.edu/pub/multimedia/mpeg/docs/ MPEG-FAQ, August 1994.

[3] MPEG-1 Standard (ISO/IEC International Standard 11172-2).

[4] John K. Ousterhout. "Tcl and the TK Toolkit," Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[5] Tom Pfeifer, et. al. "mpeg_stat (Version 1.0)," software package, ftp://ftp.cs.tu-berlin.de, 1993.

[6] L. A. Rowe, et. al. "Berkeley MPEG Video Tools (Version 1.0r1)," software package, ftp://mm-ftp.cs.berkeley.edu/pub/multimedia/mpeg/bmt1r1.tar.gz.

[7] L. A. Rowe, K. Patel, and B. Smith. "Continuous Media Toolkit (Version 3.0a2)," software package, ftp://mm-ftp.cs.berkeley.edu /pub/multimedia/cmt, March 1995.

[8] L.A. Rowe, K.Patel and B.C. Smith. "Performance of a Software MPEG Video Decoder," Proc. ACM Multimedia 93, Anaheim, CA, August 1993.

[9] L. A. Rowe, K. Patel, B.C. Smith, and K. Liu. "MPEG Video in Software: Representation, Transmission and Playback," Proc. of IS&T/SPIE 1994 Int'l Symp. on Elec. Imaging: Science and Technology, San Jose, CA, February 1994.

TABLE 2. Breakdown of callback costs

number of statistics requested	avg. time for entire callback (msec.)	avg. time to copy statistics (msec.)	avg. time to interpret assembled command (msec.)
0	4.738	0.012	0.762
1	7.681	0.958	2.737
3	20.810	4.496	11.672
6	36.532	9.738	21.614

These figures indicate that there is a base cost of roughly 4-5 msecs. to handle callbacks, with an additional cost of 4-5 msecs. for each statistic requested by the callback. This additional per-statistic cost seems to arise almost entirely from the time needed to copy arguments and, more importantly, the time needed to evaluate the assembled line of Tcl/Tk code.

Unfortunately, there is little in these figures to indicate how we might speed things up. The biggest slowdown seems to come from copying and parsing required to make the statistics collected during decoding available to the Tcl/Tk interpreter. It is not immediately obvious how these expensive steps can be avoided. An alternative to the current approach, in which all requested statistics are bundled into a single large Tcl/Tk command, would store the collected statistics in a known location and at display time, send a single one-word callback to the Tcl/Tk interpreter. This callback function accesses the collected statistics one-by-one through an extension of the play object. This approach avoids assembling the command, but in exchange requires a function call and small copy for each individual statistic used. Instead of one slow C to Tcl/Tk call passing hundreds of statistics, you have hundreds of faster Tcl/Tk to C calls each copying and passing a single statistic. It seems unlikely that this trade-off offers significant improvement in speed.

Fortunately, the applications that will be developed with these extensions are not particularly time-critical. In standard video playback, holding to the target frame rate is a key concern. In an interactive MPEG-1 analysis application, however, playback is synchronized with the presentation of detailed, macroblock-level information, and the user of the application is more concerned with understanding and interpreting this statistical feedback than with the attributes of the video playback. Under these circumstances, the user is likely to want to move from frame to frame rather slowly, taking time to glance at the feedback for each frame and perhaps even stopping for several seconds to study the attributes of a particular frame.

5.0 Discussion and Future Directions

This section summarizes the current state of these projects and discusses areas for further research.

As of this writing, the CMT extensions are complete and functional, but they have not been used to design any actual analysis tools. The obvious place to start here is to port **mpeg_blocks** and **mpeg_bits** to this new framework and to expand the interface to take

The play object was instrumented to measure these two facets of the callback process, and the code was tested on scripts with callbacks requesting 0, 1, 3, and 6 different statistics. The results are presented in Table 2.

- Extended CMT with the play object configured to make a single callback at display time. The callback requests all six available statistics, but is essentially a no-op.
- Extended CMT with the play object configured to make no callbacks at all.
- Standard CMT without the statistical analysis extensions.

For each configuration we ran a simple test which single-steps through the first seven frames of *pingpong.medqual.mpeg*, which consists of one I frame, two P frames and four B frame s, measuring the average time to decode a macroblock and the average time to handle the callbacks (i.e. to figure out whether callbacks are requested and perform them). The results are presented in Table 1.

TABLE 1. Overall cost of statistics-gathering CMT extensions

configuration	avg. time to decode macroblock (msec.)	avg. time to handle callbacks (msec.)
Extended with callback	3.703	36.532
Extended, no callback	3.342	(negligible)
Not extended	3.285	(negligible)

From the first column, it seems that testing for and servicing statistics requests while decoding a macroblock is not prohibitively expensive - collecting all six statistics slows down macroblock parsing by 13% over the non-extended CMT, while testing without collecting anything slows down only 1.7%. The display figures are more troubling. There is virtually no cost at display time for the extended CMT when no callbacks are requested (a single comparison indicates that there is nothing to do), but the cost for a callback requesting all six statistics is quite steep. As most video streams are meant to be played somewhere in the neighborhood of 30 frames per second, we should optimally spend no more than roughly 33 msecs. handling each frame. Handling a single callback adds an extra 37 msecs. to the total time required to parse and display each frame; this is clearly a problem.

We identified two possible reasons that callback management is so slow. First, when the command to be interpreted is created, all collected statistics used by that callback must be copied into a single buffer, along with the callback function name, which is then passed to the interpreter. Since each statistic list might be fairly large (i.e., size of statistics for one block x possibly hundreds of blocks.), all this copying might be expensive. Second, the call to the interpreter itself might be very expensive. Any call from C to the interpreter might be expected to be slow since it crosses several boundaries of domain, and since Tcl/Tk is interpreted as opposed to compiled. A very long command, which we have in this case, only worsens the problem since interpreting the command requires copying and parsing it.

```
$mfile ready
# set time to 1 - plays at standard rate.
$ltime configure -speed 0.1
```

Figure 8. A sample script using CMT extensions for MPEG-1 analysis.

Notice that the program only requires 22 lines of actual code. Of course, the “DisplayEvent” function will typically do something more elegant than printing to standard output. It will probably modify Tcl/Tk widgets to represent the statistics with graphics or specially formatted text.

These extensions not only simplify the initial construction of analysis applications; they also greatly simplify integration of distinct, already-completed tools. This modular design of user interfaces was envisioned by Ousterhout when he designed Tcl/Tk [4]. If several separate interfaces are designed using these extensions, combining them into a single unified tool is as simple as configuring a single play object to invoke all of the callback functions used in all of the different applications.

Using CMT also frees the designer from having to “re-invent the wheel” when it comes to flow control and other basic video management issues. Whereas in **mpeg_blocks** and **mpeg_bits** we had to put considerable effort into supporting limited VCR-like commands, a designer using the extended CMT gets flow control “for free.” The timer object in cooperation with the index generated by the file object reduces the problem of arbitrary speed and positioning to a simple one-line configuration command.

Finally, the underlying data structures and procedures used to implement these extensions were designed with the understanding that users may want to collect statistics beyond those provided. We therefore designed with an eye on extensibility, and we believe that we have succeeded. Originally, the extensions to the play object were designed to support only four statistics (i.e., “sizemb”, “typemb”, “motionvector” and “dctsnumber”). Modifying the code to support “cbppattern” and “sliceloc” took one person on the order of 30 - 40 minutes, most of which was spent trying to find the best place in the decoding algorithm to collect the respective statistics.

4.4 Performance Issues

These extensions involved modifying the software decoder object used to parse each block of each frame, as well as the play object used to actually present each frame once decoded. The decoder modifications involved testing when a macroblock is decoded which statistics are being collected, if any, and writing to a buffer devoted to statistics of that particular type. The play object modifications involve assembling the Tcl/Tk callback from the given function name and the collected statistics, then evaluating the assembled command in the Tcl interpreter.

To get some idea of the expense of these modifications, we instrumented the objects to record their speed in performing key tasks. We investigated three configurations:

callback should also query the result of getStats to get the size and type of the current frame, and use this information to fill in the scrolling frame and bitrate widgets. Finally, we use the play object's "configure" option to request that the callback be called at every frame display. The entire interface can be coded in a Tcl/Tk script; the application designer does not need to look at any of the decoding algorithms or object definitions (unless he wants to collect a currently unsupported statistic).

Figure 8 shows a simple script that plays the video stream *pingpong.medqual.mpg*, printing to standard output each time a frame is displayed the type of the frame and the average size of a macroblock in the current frame.

```
# a simple video playback with CMT
wm minsize . 1 1
# initialization
set fileStatList {}
set numBlocks 0
# function called at each frame display. frameNo is the number of the
# frame to be displayed, and sizeList is a list of the macroblock sizes
# for that frame.
proc DisplayEvent {frameNo sizeList} {
 global fileStatList
 global numBlocks
 set totalSize 0
 # print current frame's type
 puts "type: [lindex [lindex [lindex $fileStatList $frameNo] 5] 1]"
 # now output the average size
 for {set i 0} {$i < $numBlocks} {incr i} {
  set totalSize [expr $totalSize + [lindex $sizeList $i]]
 }
 puts "avg. mb size: [expr $totalSize/$numBlocks]"
}
# create the timer, play and file objects
set ltime [lts ""]
set mfile [mpegFile ""]
set mplay [mpegPlay ""]
# connect file object to source file, player, and timer
$mfile conf -lts $ltime -outCmd "$mplay accept " -filename /n/vid2/data/
mpeg/movies/pingpong.medqual.mpeg
# load the whole stream into the file object
$mfile addSeg 0 end
# get the list of stream and frame level stats
set fileStatList [$mfile getStats]
# find number of blocks in the display from display width and height
set numBlocks [expr ([lindex $fileStatList 1] * [lindex $fileStatList
2])/256]
# connect the player to the timer and a window
# instruct play object to collect macroblock size info and
# pass it at display time to DisplayEvent
$mplay conf -lts $ltime -xid [winfo id .f] -callbacks {{DisplayEvent
sizemb}}
# note that everyone is ready
$mplay ready
```

where “size” and “type” give the size in bytes and type (e.g. I, P or B) of the corresponding frame and “time” gives the time when the frame should be displayed.

For the play object, we added a new configuration option, “-callbacks.” Given a play object “mplay,” the command

```
$mplay configure -callbacks {{func1 stat1_1 stat1_2...} {func2 stat2_1
stat2_2...}...}
```

configures the play object to make callbacks to the given functions at frame display time, using as arguments to the function the statistics listed. Each “func*i*” must be a Tcl/Tk procedure of the form

```
proc funci {frameNo arg1 arg2....}
```

When frame *n* is displayed, each listed func*i* will be called with its “frameNo” argument set to *i*, and “arg*j*” equal to the statistics list corresponding to stat*i_j* for frame *n*. Currently available statistics are:

- sizemb: a list of the sizes in bytes of each macroblock in the frame.
- typemb: a list of the type of each macroblock in the frame, where the type could be skipped, intra-coded, forward predicted, backward-predicted, or bidirectional.
- motionvector: a list of the forward and backward motion vectors for each macroblock in the frame, supplying in each case the coordinates of the vector or indicating that the vector was not used.
- dsctsnumber: a list of the number of non-zero DCT coefficients used to code each macroblock in the frame. Each entry in the list is a list of six integers, giving the number of non-zero DCT coefficients used to encode each of the six blocks in the macroblock.
- cbppattern: a list of the coded block patterns for each macroblock in the frame. A coded block pattern is a six-bit string where the nth bit being set implies that the nth block has non-zero DCT coefficients.
- sliceloc: a list of slice beginnings in the frame. Each number in the list gives the address of the first non-skipped macroblock in the frame.

In all of the above, macroblocks are ordered from left to right, top down. So the first entry in the “size” list gives the size of the top left macroblock, the second entry corresponds to the macroblock just to its left, and so forth.

4.3 Evaluation

The functionality supported by these CMT extensions simplifies the implementation of interactive MPEG-1 analysis tools. The designer need only lay out his interface; the CMT extensions take care of collecting and synchronizing the presentation of any desired statistics. To re-implement **mpeg_bits**, for example, we can use “getStats” to collect stream-level information on the size of the picture, the target bitrate, etc. and use this information to lay out the main information window. Next, we write a callback function that takes a list of block sizes and uses that information to color in the bitfield. The

4.1 The Continous Media Toolkit (CMT)

For the purposes of this paper, I will discuss very briefly three CMT objects: 1.) a "file" object, 2.) a "play" object, and 3.) a "timer" object. For a more complete description see CMT documentation [7]. These objects can be thought of as components one uses in watching a video at home on a VCR. The "file" object corresponds to the tape itself. The "play" object corresponds to the internals of the VCR and the TV set. Given a frame of MPEG-1 encoded video, a play object decodes and displays the frame. The "timer" object corresponds to the controls on the front of the VCR. It tracks the current time value and determines which frame is currently being sent from the file object to the play object. The speed variable in a timer object determines when the next frame (or previous frame, if speed is negative) will be sent.

File and timer objects offer some key elements of stream control unavailable in **mpeg_play**. When a segment of MPEG-1 video is added to a file object through the addSeg command, it is immediately parsed to generate an index to the frames in the segment. This index contains pointers to the start of each frame, the size and type of each frame, a pointer to any frames that the frame uses as reference frames, and the time when the frame should be played if the video is played at normal speed. The timer object, given this index, has all the information it needs to display frames, in any order, at any time. Given the current time value, a timer object uses the index to select the information needed to decode the frame for that time value, and triggers the file object to send it across to the play object. The play object takes this information and decodes the frame into distinct "frame" objects that can then be played or used as reference frames.

4.2 Extensions to CMT for Interactive Analysis Tools

The goal of this project was to extend CMT so that users can quickly and easily design and configure their own MPEG-1 analysis tools. The key idea was to handle the complexities of gathering statistics and synchronizing presentation and display within the object definition, freeing the application designer from the data handling and control details so he can focus exclusively on the layout and behavior of the interface.

This goal was accomplished by modifying the file and play objects to provide statistical feedback on request. For the file object, a new command "getStats" was implemented. Given a file object "mfile," the command

```
$mfile getStats
```

returns a list of statistics on all currently loaded segments of the form

```
{numFrames width height bitrate secPerFrame frameList}
```

"numFrames" gives the total number of frames, "width" and "height" give the dimensions of the display in pixels, "bitrate" indicates the target bitrate and "secPerFrame" gives the expected display rate. "frameList" is a list with one entry for each frame, an entry being of the form

```
{size type time}
```

4.0 CMT Extensions

Our experience designing **mpeg_stat**, **mpeg_blocks**, and **mpeg_bits** gave us several useful insights about the task of designing analysis tools for MPEG-1 streams:

- It is hard to satisfy everyone with one tool. **mpeg_stat** collects just about every bit of information that can be gathered, but it does not present it in a simple, easy to understand visual format. There is simply too much information. **mpeg_stat** also trades speed for visibility. It gathers statistics much faster than either of the other two tools because it does not display the actual video, but this approach denies the user a level of understanding and insight that he can get from being able to view images and corresponding statistics side-by-side. **mpeg_blocks** and **mpeg_bits**, on the other hand, provide intuitive graphical feedback synchronized with video playback. However, they are very slow and limited in focus; each only presents information on a narrow subset of the available information.
- **mpeg_play** was a poor basis for developing interactive tools. **mpeg_play** is optimized for speed and simplicity. A truly useful interactive tool for MPEG-1 analysis should support random access, and it should handle very long video streams. These two goals together are rather difficult to build into **mpeg_play** without seriously restructuring the whole program.
- The technically most difficult aspect of building a GUI interface like **mpeg_bits** or **mpeg_blocks** is gathering and displaying the statistics. For both applications, the layout and behavior of the GUI interface had be carefully designed to be intuitive and informative, but once designed it was not particularly difficult to implement. The most difficult and time consuming part of designing these applications was modifying the video-playback code. Where should we look to compute a particular statistic? Where should it be stored, and when should it be displayed? How do these answers change in light of the possibility of moving backwards through the video stream, or jumping to random locations?
- Any tools that give feedback on encoding in synchronization with video playback will be pretty similar on the video decoder side. While the questions of where to find statistics, how to store them, when to display them, and how to control the video stream, are initially troublesome, once answered these solutions are pretty much the same no matter what statistic or statistics your application will be presenting, whether it is size, bitrate, DCT coefficients, block type, or motion vectors.

These observations suggested that we investigate building a more general, configurable interactive analysis tool on a more sophisticated video playback system. The Continuous Media Toolkit (CMT) is an attempt to simplify the design of networked multimedia systems by creating "objects," with predefined properties and methods that can be instantiated, linked together and used to construct applications. The remainder of this section provides a brief overview of CMT and describes the design and implementation of extensions to CMT that support construction of a variety of MPEG-1 analysis tools.

This approach risks violating the first expectation of atomicity, in that there is a long period of "uncertainty" between the beginning of the edit and its confirmation. We counter this uncertainty by clearly indicating at each phase of the edit that the edit will have no effect until it is completely specified and confirmed. A clearly marked "Cancel" button is available at each phase of the edit, and invalid inputs generate a warning and prompt the user to try again. In general, the problem is not as severe as it is in other editors because most editors support a number of different editing operations: cutting, pasting, changing font, changing document layout, and so forth. The real problem with non-atomic edit operations is how to interpret overlapping edits. What exactly should happen when I get halfway through a delete, then add new text, paste more text into the document, complete the delete, and then perform another paste operation? **mpeg_bits** does not encounter these problems because there is only one edit operation - specifying q-scale changes across regions of the video - and the application enforces that only one of these operations may be ongoing at a time.

The problem of making local changes was handled by avoiding the issue altogether. Instead of changing the actual video stream, **mpeg_bits** generates a file to re-code the stream according to the parameters specified. The Berkeley MPEG-1 encoder automatically handles the problem of propagating changes to reference frames to dependent frames, leaving **mpeg_bits** free to manage changes as isolated items.

The problem with this solution is that it seems to violate key expectations of editor interfaces listed. The most serious shortcoming is that changes are not viewed immediately; if the user inputs an edit to improve the quality of a range of blocks across a certain range of frames, the frames look exactly the same after the user confirms the edit. Along the same lines, the Save/Restore notion is not supported as expected. A user cannot open an MPEG-1 file with **mpeg_bits**, specify some edits, save his work and exit, and then re-open the same file to see the stream changed according to his previous work.

Our answer to these charges is to make it clear that **mpeg_bits** is a CDL editor, not a video editor. Once an edit is confirmed, it is immediately visible in the Edit List, where it can be deleted, viewed in detail, or otherwise modified. At any time a user can save the CDL he has generated, and he can later reload that file and continue to edit it.

The confusion arises because the CDL file is invisible during an **mpeg_bits** session, while the video and associated controls are very prominent in the layout of the interface. This invisibility is not necessarily a problem, however, in that the whole selling point of **mpeg_bits** is to generate automatically a CDL file without having to interact directly with the file itself. CDL files tend to be hundreds if not thousands of lines of rather undescriptive, formatted commands that are very difficult to interpret or to code by hand. **mpeg_bits** provides a more intuitive, graphical interface to the CDL. Nevertheless, a future project might build a fully interactive MPEG video editor.

Designing an interface for users to specify compression decisions, however, was a more difficult task. Most editors that we see are for 2-dimensional mediums: text editors, paint programs, and so forth. The interfaces to these editors tend to have a few expected "rules" of behavior, namely:

- Atomic changes. Users expect edits to be binary, that is, either the edit happens or not, and the edited object is never left in some in-between state. For example, if the user moves a paragraph from one location to another in a text editor, he should never see or interact with a state where the paragraph is at both locations at once.
- Immediate feedback. Users expect to see the effects of their changes immediately. In a paint program, for example, if a user changes a square's color to red, it is not acceptable to assure the user that the change has taken effect and then actually change the color five minutes later.
- Save/Reload. Users expect to be able to stop an editing session at any time and save their work. Later they can restart the editor, reload their work, and continue where they left off.
- Undo. Edits, especially just-made edits, should be undo-able with a simple, quick operation.

We wanted **mpeg_bits** to meet as many of these expectations as possible, but there were two major complications, both relating to the nature of the object being edited. First, video is a 3-dimensional object. It has height, width, and time. Since both the computer terminal and the mouse are 2-dimensional I/O devices, this limitation presents a problem. How does the user quickly and easily specify an edit that may span three directions when he can only give and receive feedback in two dimensions? Second, MPEG-1 compression is complicated, in the sense that how a given block or frame is encoded, depends heavily on the contents and compression of the other reference blocks and frames in the stream. Therefore, changing the encoding of a single macroblock in a single frame is not as simple as replacing the bits that previously represented the macroblock with a new set of bits. Any block encoded using a motion vector pointing to pixels in the changed block must be re-coded, since these pixels have changed. In fact, even if a block MIGHT have been encoded using the changed pixels, that block needs to be re-coded because the choice of reference block might change given that some pixels in the reference frame have changed. This problem is recursive. Each block changed as a result of the first change in turn requires that any potentially dependent blocks be changed. Since P frames can depend on each other, it is possible that a single change to the encoding of a single macroblock might propagate changes to many other frames.

The first constraint is handled in **mpeg_bits** with the use of an "edit" mode. The user specifies the height and width of the regions he wants to change within a single frame, enters "edit" mode by selecting "Mark First Frame" from the Edit menu. At that point he specifies the frames he wants to include in the edit (i.e. the "time" dimension). Once the dimensions of the region to change are specified, the nature of the change is entered, the edit is confirmed, and the application leaves "edit" mode.

frame after that, and any frames in between the two are the next frames we'll encounter in the stream.

So, to support rewind (or large-step-back, as it was implemented), **mpeg_play** was modified so that when a GOP header is encountered during parsing, a state holder, with the current file offset, is created and added to a list associated with the vidStream object. Then when a rewind is requested, all position-related state is be cleared from the vidStream object: variables indicating bit, byte, and file offsets, the vidStream's "chunk" buffer, past and future frame buffers, etc. The file offset is set to that of the last saved state, the other offsets are set to 0, the "chunk" buffer is filled according to the new file offset, and the reference frame holders are left empty. At this point **mpeg_play** has all the information it start displaying frames from the restored location.

3.2 Synchronization of Feedback

As discussed above, many frames are displayed some time after they are parsed and decoded. For **mpeg_blocks** and **mpeg_bits**, this presented a problem, in that the statistical feedback displayed in the information window should obviously be describing the frame currently displayed, but almost all information is collected while parsing of the stream. We needed to instrument the **mpeg_play** code to collect the statistics during parsing, store it somewhere and then display it when the frame is actually displayed.

This problem was solved by designing a set of statistics buffers that are managed in parallel with the frame buffers. As **mpeg_play** decodes a frame, the image is reconstructed in a frame buffer. When the frame is completed and the display procedure is called, if the frame is an I or P frame, the decoded image is saved in a "future buffer" slot while the previous contents of this slot are displayed, as described above. Under our modifications, the decoding procedures also write statistical information to a statistics buffer, creating what is essentially a Tcl/Tk script to configure the information window to display the relevant statistics for the frame being decoded. In oher words, the statistics buffer conatins Tcl/Tk code that updates the information windows. When the display procedure is executed, the statistics buffer is processed the same way the frame buffer is processed: if the frame is an I or P frame, the statistics buffer waits in the "future script" variable while the previous contents are executed. Otherwise, if it is a B frame, the frame is displayed immediately and the script is executed immediately. Unlike frame buffers, there is no "past script" buffer because once a script is executed it is not referenced again.

3.3 User Interface Issues

The key issue in designing **mpeg_bits** was to create a useful, intuitive interface for viewing and modifying the bit allocation patterns of an MPEG-1 stream. The visual feedback aspect was not too difficult. Objects such as macroblocks or frames are represented as rectangles, with the layout and location of each rectangle indicating the actual item to which it corresponds, with differences in size and color highlighting the different properties of each item.

progress through the stream depends on some user-controlled variable. This variable can be set to halt progress once a new frame has been parsed and displayed. Changing the input midway through processing (i.e. opening a new video stream without stopping and re-starting the whole application) was only slightly more difficult. The main challenge was to catch all of the different variables and states that needed to be reset and reconfigured for the new stream, including display size, file and buffer offsets, and reference frame buffers.

The biggest challenge in control was “fast access.” If the user knows he wants to get to frame N, how can he get there as quickly as possible, no matter where he is in the stream? The optimal solution is to supply an interface with which specific frames can be requested, and support random access. For the reasons mentioned above, this approach is very difficult, although other researchers at Berkeley have implemented this function [9]. Knowing the display number of a frame does not tell you the order in which the frame appears in the stream and, even if you knew the stream order of the frames, you do not know the offset of the frame or the offset of its reference frames.

Alternatively, fast access can be supported by fast forward and rewind buttons. The problem with fast forward, as with random access, is that you cannot simply jump forward in the stream. To move forward, the stream must be parsed. To move forward N times faster, the application must parse the stream N times faster, which is difficult considering that **mpeg_play** is already optimized for speed. On current processors, **mpeg_play** can dsiplay 160x120 movies faster than real-time, but 320x240 movies are displayed at approximately 16 frames per second. Without a table of offsets and dependencies as described above, the only way to support fast forward is to speed up parsing, for example by displaying frames selectively or not at all.

A rewind button, however, is more plausible, since rewind involves going back to a state previously held by the playback application. The key question becomes, what is the least amount of state the application needs to retain in order to get back to previous frames? The solution we supported involves selectively saving state at reasonable points in the stream, when dependency relationships are simple and easy to reconstruct. For these tools, we save state at a “GOP header.” A GOP header is a good placeto checkpoint because they are frequent but not too frequent, the dependency relationships are fairly clear, and we know that any special decoding instructions for the following frames will be given in the header.

Given these conditions at the start of a GOP header, the only state we need to worry about is the offset into the stream at the start of the GOP header and the contents of the future and past frame buffers. Saving these buffers, however, is very space-inefficient, because the application will typically checkpoint every 10-20 frames, and a decoded frame is a fairly large data item. It turns out, in fact, that saving these reference frames is not really necessary. Starting with two empty reference buffers, P and B frames will be discarded until an I frame fills the future buffer. At that point B frames will be discarded until an I or P frame is discovered, which will replace the I frame in the future buffer, display the I frame and move it to the past buffer. Now every thing is synchronized: the past buffer contains the last reference frame played, the future buffer contains the next reference

3.1 Adding Flow Control to mpeg_play

The approach to video playback used by **mpeg_play** is essentially as follows: a "vidStream" object is created that is composed of a large buffer with bit and byte offsets and several variables that contain different attributes of the video stream. The MPEG-1 file is opened, and a buffer-sized chunk is read into the vidStream object and parsed according to the MPEG-1 standard. When the buffer is nearly exhausted, the next chunk is read from the file and processed.

At the beginning of an MPEG-1 file is a stream header that contains various stream attributes such as display size, frame rate, and target bitrate. These are used to set variables in the vidStream object. Once this information is processed, **mpeg_play** enters a parsing loop that parses off various headers and blocks until an entire frame can be reconstructed. It is important to note two facts that complicate the control of playback for an MPEG-1 stream. First,compressed frames, macroblocks, and blocks are of variable size because the compression factor of MPEG-1 encoding varies depending on the input and other user-defined parameters. Therefore, there is no way to jump forward in the stream to the beginning of a certain block or frame, and there is no way to make sense of a particular offset in an MPEG-1 file without all of the state that would be gathered by parsing the stream from the beginning. Second, frames in an MPEG-1 file are specified in transmission order, not display order because B frames must be decoded in terms of frames that are displayed later in the stream. Instead, reference frames must appear before frames that refer to them. This constraint has a ripple effect, so that I and P frames appear several frames ahead of their display position. For example, suppose the frame display order is as follows:

I_1 B_2 B_3 P_4 B_5 B_6 I_7 B_8 B_9 P_{10} B_{11} B_{12} .....

These frames are stored in the MPEG-1 file as follows:

I_1 P_4 B_2 B_3 I_7 B_5 B_6 P_{10} B_8 B_9 I_{13} B_{11} B_{12} ...

to simplify encoding a file and sending it to a remote client to be decoded.

To account for this re-ordering, **mpeg_play** decodes an I frame and saves it in a "future frame" buffer, while the previous contents of the future frame buffer, if any, are displayed and moved into the "past frame" buffer. When a P frame is encountered, it is decoded using the contents of the past frame buffer as a reference frame. Once decoded, as with an I frame, the P frame goes into the future frame buffer, the previous contents of which are displayed and moved into the past frame buffer. When a B frame is encountered, it is decoded and displayed immediately. While at first this seems rather complex, with careful thought it becomes clear that in any possible pattern this algorithm handles the re-arrangement of frames, presenting them in order and providing each frame with the proper reference frames, where needed.

Given these properties of an MPEG-1 stream, it is easy to see the flow-control problem. Pausing and stepping forward are simple: we simply installed breaks and flow-control variables into the main parsing loop so that instead of continually parsing and displaying,

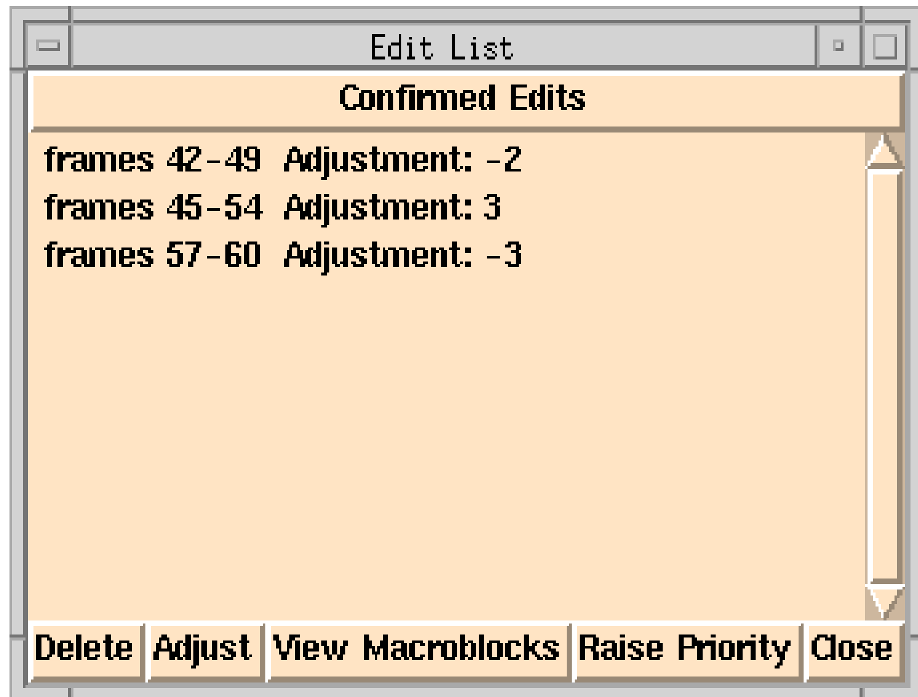

Figure 7. The Edit List.

3.0 Implementation Issues

This section describes some of the more difficult technical challenges we faces in implementing these tools. As mentioned above, **mpeg_stat**, **mpeg_blocks** and **mpeg_bits** were implemented by modifying **mpeg_play**. These modifications can be loosely grouped into four categories:

- Control. **mpeg_play** simply opens and plays an MPEG-1 video file, without supporting an interface to control the playback. Since both **mpeg_blocks** and **mpeg_bits** provide information about the currently displayed frame, both tools must allow the user to position the video stream on any arbitrary frame.
- Feedback. For each tool, **mpeg_play** had to be instrumented to collect the required statistics at the proper point in the decoding process, and to somehow format and display these statistics in a meaningful way.
- User Interface. **mpeg_bits** enables the user to design specifications for future encodings. Since **mpeg_play** does not support an interactive user interface, we had to design and implement a workable, intuitive interface for editing the characteristics of a video stream, within individual frames and across a series of frames.

The remainder of this section details several specific approaches that were used in supporting these modifications.

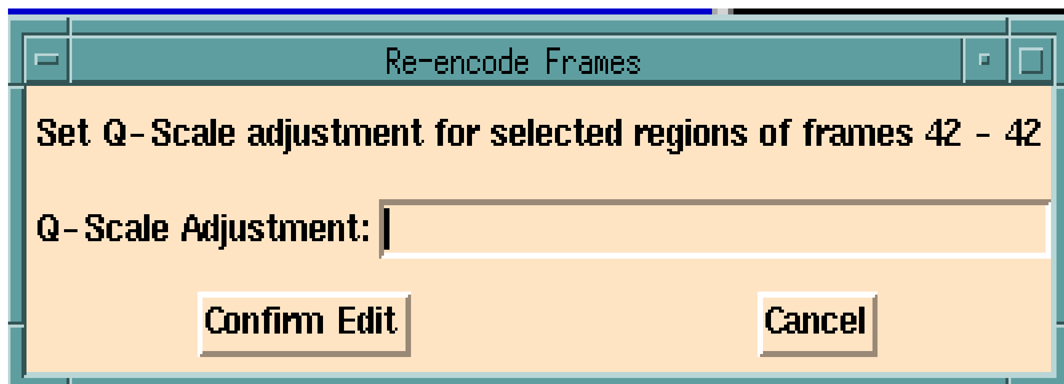

Figure 6. The Re-encode Frames dialog.

During an **mpeg_bits** session, the user may wish to view or modify the edits he currently has confirmed. To do so, he selects "View Edit List" from the Edit menu, which produces the "Edit List" dialog (Figure 7). The list displays, in order of priority, the edits confirmed for this session, listing the frames involved and the q-scale adjustment requested. The priority of an edit simply means that if both edits involve changes to the same macroblock on the same frame, only the edit with the higher priority will be performed. As Figure 7 indicates, the Edit List allows the user to delete an edit, adjust an edit's frame range and q-scale shift, raise an edit's priority to switch places with the one above it, or view the pattern of selected macroblocks associated with an edit.

Once the user is satisfied with the edits he has made, he may use the Save command in the File menu to save his work to a compression decision list (CDL). This file can be used to re-encode the stream according to the specified edits using the Berkeley MPEG-1 Software Encoder [1, 6]. This encoder is configured by a user-defined parameter file, which is used to modify the MPEG-1 encoding discussed above, including IPB pattern, motion vector search pattern, and quantization factors, among other things. To re-encode using the **mpeg_bits**-generated file, the user simply modifies the parameter file to include the line:

```
CDL_FILE cdl_file_name
```

Once the stream is re-encoded, the user may use **mpeg_bits** again to verify that his changes performed as desired. In this second session, he may reload the edits generated by the previous session using the -edits flag on the command line or the Open Edit List option from the File menu. This command automatically reloads the previous edits into the confirmed Edit List, where they can, as before, be deleted or modified at the same time new edits may be defined.

mpeg_bits also provides an interface to modify the encoding of a video stream. Suppose that as a certain video stream is being analyzed using **mpeg_bits**, the user sees that a certain image (say, someone's face) is not coming out very clearly. He refers to the "bit field" and sees that for whatever reason, relatively few bits are being allocated to that particular area in frames 45 - 65. Looking at the bitrate gauge, he sees that he is fairly close to the target bitrate and therefore does not want to re-encode the entire stream at a lower quantization factor, which would result in more bits being used to encode each block. Instead, he can use **mpeg_bits** to describe a localized edit for use in the next encoding.

Using the control panel, he positions the stream to the first frame to be included in the edit, frame 45. Using the left mouse button, he selects which macroblocks he wants to include in the edit using a selection box (i.e., position mouse cursor, mouse button down, drag to new position, release mouse button) on the "bit field" or on the video display itself. The selected macroblocks will appear shaded on the "bit field". If at any time before the edit is confirmed the user wishes to change which blocks are selected, he may add more blocks with additional selections , or he may click the right mouse button to deselect all selected blocks.

Moving to the Edit menu, the user can select "Mark First Frame," which produces the "Frame Selection" dialog. This dialog is the first in a series of dialogs that will guide him through the edit (Figure 5). He then positions the stream to the last frame in the edit, in this case frame 65, and selects "Mark Last Frame." This brings up the "Re-Encode Frames" dialog box (Figure 6), which prompts the user for a positive or negative q-scale adjustment which is to be applied to the selected blocks in the selected range of frames. Once the edit is confirmed, the dialogs disappear and the selected blocks no longer appear shaded.

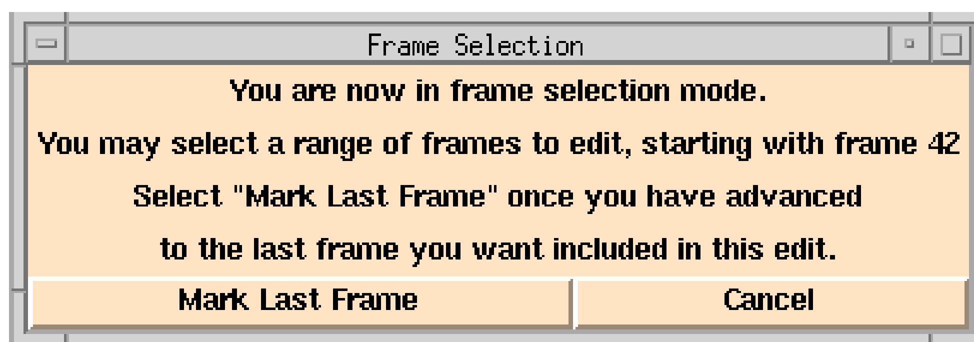

Figure 5. The Frame Selection dialog.

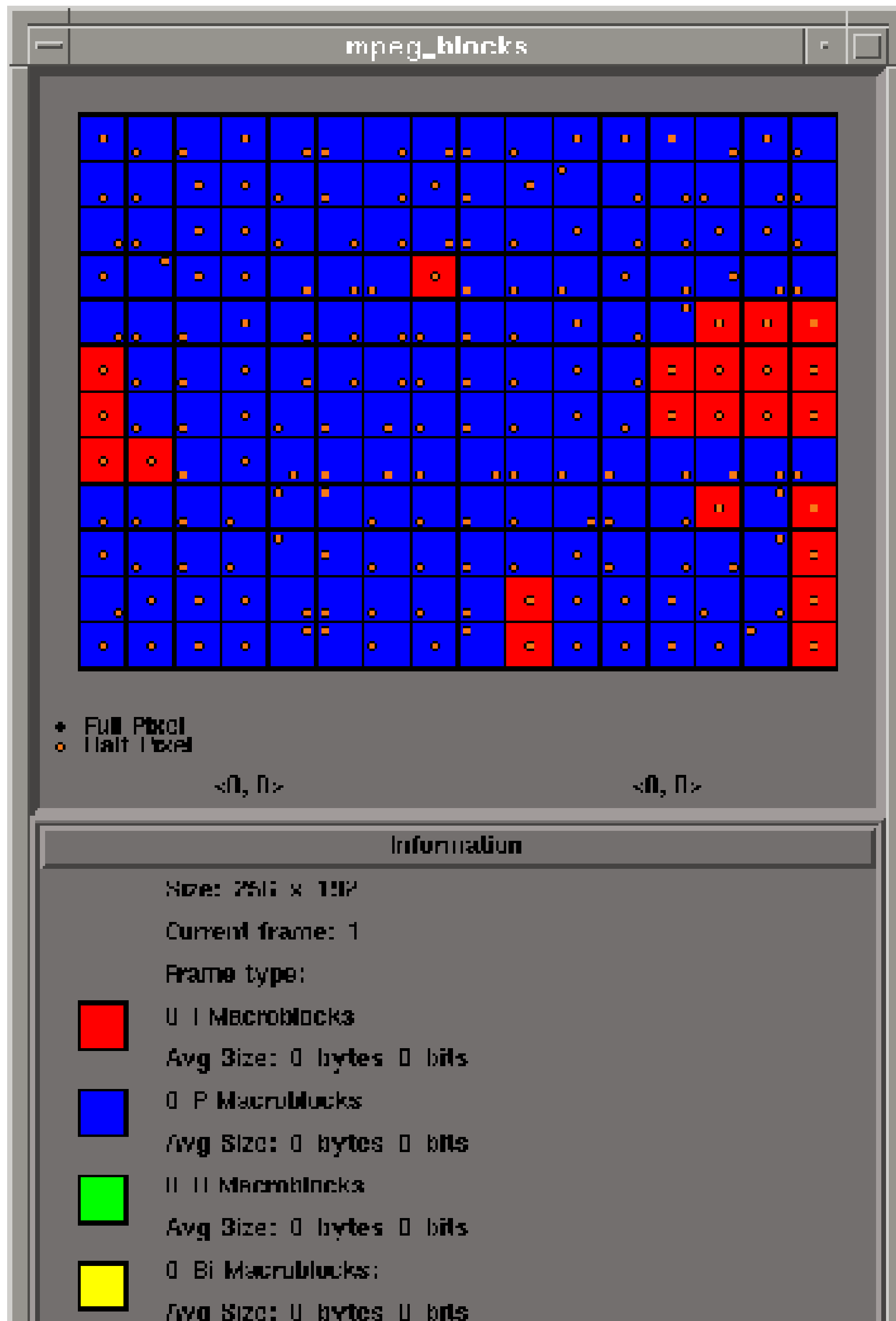

Figure 4. The information window for mpeg_bits.

2.3 mpeg_bits

mpeg_bits is run with the command line:

```
% mpeg_bits[-edits edit_file_name] [-dither dither_option] [file_name]
```

Similar to **mpeg_blocks**, **mpeg_bits** creates three new windows: a display window, a control panel, and an information window. Here, the control window has buttons for playing and stopping, stepping forwards and stepping backwards. An example of the information window for the video stream pingpong.medqual.mpeg appears in Figure 4.

At the top of the information window is the "bit field," similar to the grid in **mpeg_blocks**. Instead of indicating block type, however, the color of each grid block indicates the size of the corresponding macroblock. A darker color indicates that more bits are used to encode the block; a lighter color indicates that fewer bits were used to encode the macroblock. The legend at the bottom of the bitfield correlates the various shades with actual block sizes. Clicking the middle mouse over a macroblock displays its exact size in bits at the top right corner of the information window.

Below the bit field is the "frame information window," which gives the current frame number, size in bits, and type. To the right of the frame information window, the "relative frame sizes window," which is a scrolling widget indicating the relative sizes and types of the past few frames. Clicking over a rectangle in the scrolling widget with the middle mouse button displays the size in bits of the corresponding frame in the top right corner of the information window.

Finally, at the bottom of the information window we have the "bitrate gauge." The bitrate gauge shows the frame rate and target bit rate of the video stream. It also provides a scrolling gauge comparing the target bitrate to the actual number of bits used over the past second.

mpeg_bits is useful for answering questions about bit allocation on a macroblock level. Watching the "bit field" at the top of the information window alongside the actual frames, users can build intuition about what kinds of images compress well and alternatively what image characteristics contribute to a less efficient compression. The macroblock-level feedback can be used to investigate how changing different parameters in the encoding, e.g. motion vector search pattern, quantization factors, frame patterns, etc., affects the way bits are distributed. **mpeg_bits** can also be used to deal with the critical issue of bitrate control. Analyzing an MPEG-1 stream with **mpeg_bits**, a compressionist can spot areas where the actual bitrate exceeds the target bitrate and actually see which frames and which particular regions of these frames are the source of the problem.

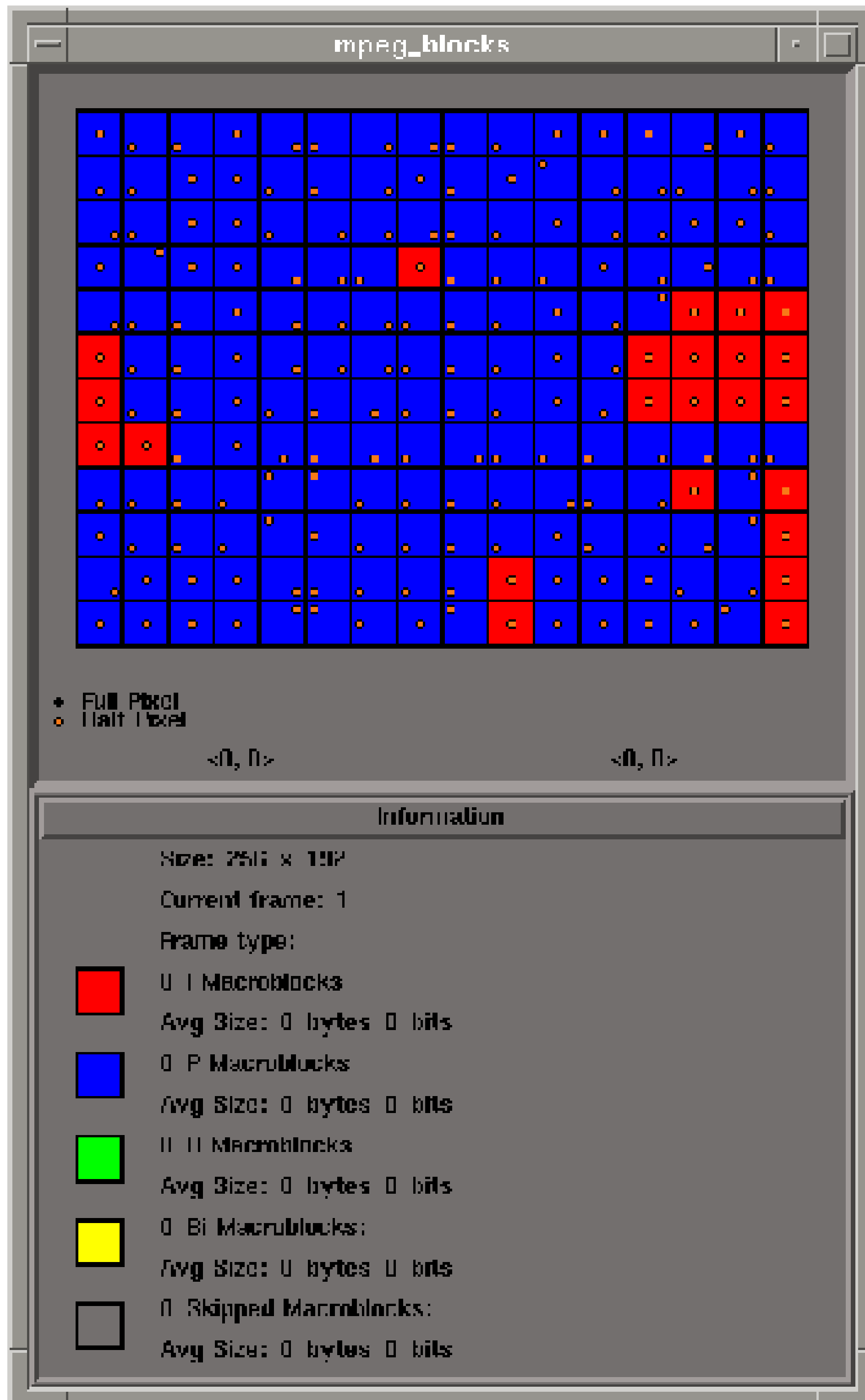

Figure 3. Information window for mpeg_blocks.

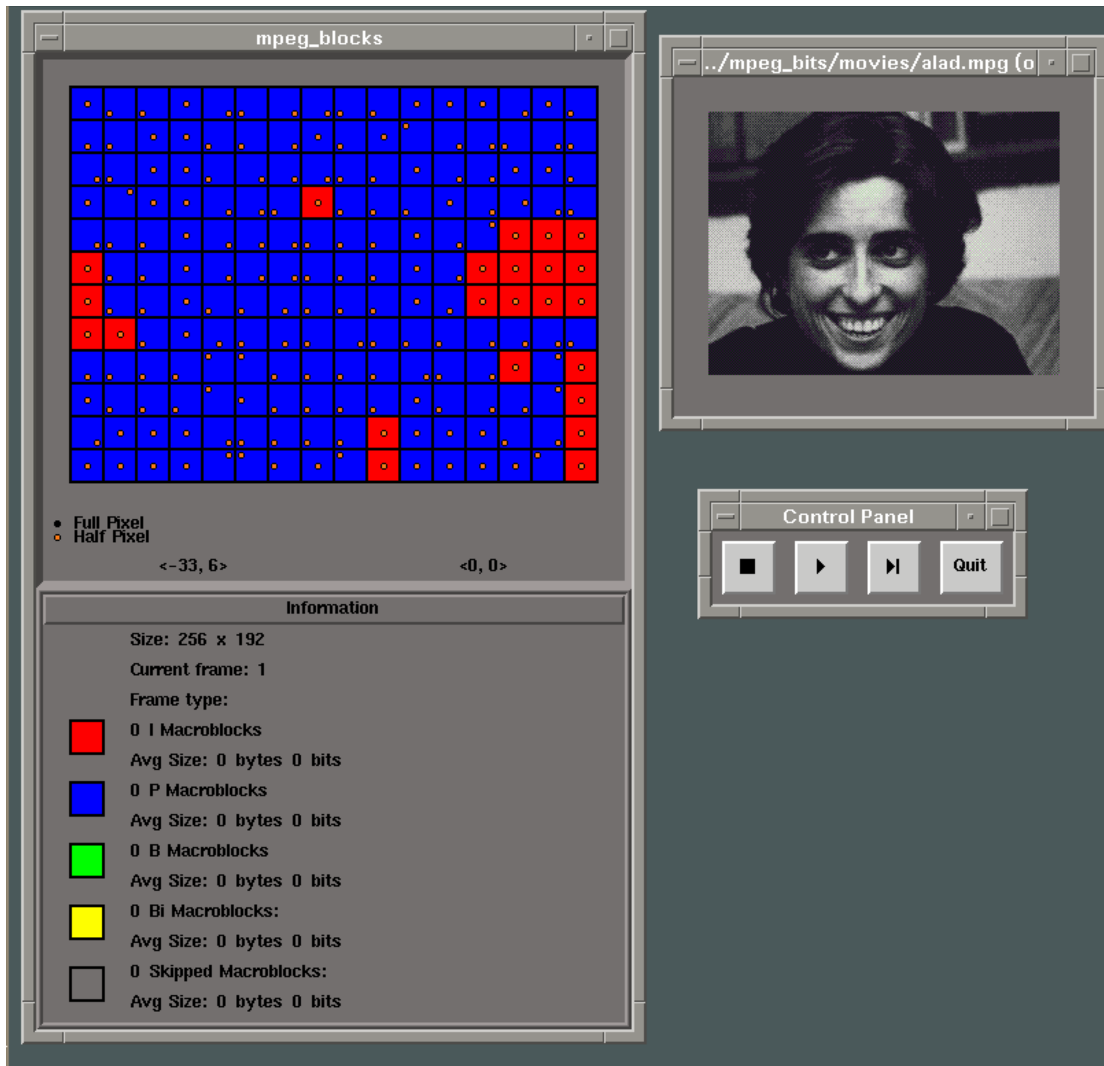

Figure 2. mpeg_blocks.

2.2 mpeg_blocks

mpeg_blocks is run with the command line:

```
% mpeg_blocks file_name
```

This tool creates three windows on the desktop: the display window, the control window, and the information window, as shown in Figure 2. The display window displays the actual video sequence. The control window contains buttons to play, stop, and step through the video stream, as well as a "Quit" button. The information window gives visual feedback on the allocation of bits within the video stream. An example of the information window for the video stream *alad.mpg* appears in Figure 3.

At the top of the information window is the macroblock grid: each square in the grid represents the macroblock in the corresponding location in the currently displayed frame. The color of the grid block indicates the type of macroblock, i.e. whether it uses motion vectors and if so how many and what direction. A macroblock can also be "skipped," which means that it was so similar to a reference macroblock that the previous macroblock can be reused.

mpeg_blocks also provides information about motion vectors. At the bottom of the grid there are two ordered pairs of integers. When the mouse is positioned over a grid block, the pair on the left gives the coordinates of the backward-pointing motion vector for the corresponding macroblock (i.e. the motion vector referring to a macroblock in a previous frame). The pair on the right gives the coordinates for the forward-pointing motion vector. When the mouse is positioned over a macroblock for which one or both of the motion vectors are undefined, the coordinates for the unused vector are given as <0,0>. In addition, a small dot in each macroblock indicates the general direction of the vector or vectors used in that block. As indicated by the legend at the lower left of the grid, the color of the dot indicates whether the motion vector is given in half- or full-pixel coordinates.

Below the grid is a series of other relevant statistics. **mpeg_blocks** provides the size of the display in pixels, the current frame number and type, and a histogram of the macroblocks in the current frame, giving the number and average size for each type of macroblock.

The information supplied by **mpeg_blocks** is useful for evaluating how a given encoding exploits temporal redundancy. Suppose, for example, a researcher wanted to evaluate the effectiveness of several different motion vector search algorithms. Using **mpeg_blocks**, he can see at a glance how successful each algorithm is at finding matching macroblocks.

- Total compression ratio for the entire stream.
- Average size, compression ratio and quantization scale for each frame type.
- Total number of skipped and coded macroblocks, given as a percent of the total number of macroblocks.
- MPEG-Viewer requirements, which lists the requirements and specifications for the encoded stream, including aspect ratio, required frame rate, target bit rate, requested buffer size. This data is retrieved from various stream headers. This section also indicates which, if any, of these targets the stream fails to meet. For example, if the bit rate ever goes above the target bitrate or the stream ever requires more buffer space than is available, these problems are reported.
- Motion vector information, including maximum and average length in horizontal and vertical directions, for forward and backward vectors.

In addition, **mpeg_stat** now supports a number of command line flags to gain very detailed feedback on a block level. In general, these options are of the form:

```
-statistic file_name
```

This collects statistics of type “statistic” into a file named “file_name.” The available statistics are:

- qscale: Q-scale and custom q-matrix information. For each type of frame, indicates the q-factor and number of non-skipped blocks of that quality.
- size: Type and size of each frame.
- offsets: Offset of each picture, group of pictures (GOP) header, and slice header in the stream. GOPs and slices are described elsewhere [1].
- rate: Bitrate at every picture, after the first second of video. Also gives a summary of the highest and lowest bitrates encountered in the stream. This data can be modified with the flag “-ratelength N,” which indicates that bitrate should be given in bits/N frames as opposed to bits/second.
- block_info: A thorough picture of each block, including number and type of containing frame, number and q-scale of containing slice, block number, block q-scale, block size, and motion vectors if there are any.
- dct: Adds the actual dct coefficients for each block to the block file.

It is also possible to collect all statistics simultaneously but record them into separate files.

mpeg_stat enables the user to collect volumes of data on many aspects of the encoding process, including bit allocation on a block-by-block basis and success in exploiting temporal locality. This comprehensive view of the entire stream is useful to anyone concerned with very low-level questions about the encoder in question, for example, someone designing and debugging a new encoder. **mpeg_stat** also explicitly answers the compressionist’s key concerns about whether or not an encoded stream meets the requirements imposed by a particular playback environment; for example, bit rate and space constraints.

2.0 The Plateau MPEG-1 Video Analysis Tools

Building on **mpeg_play** and **mpeg_stat**, the Plateau Multimedia Group has developed tools to analyze and even modify an MPEG-1 video stream. These tools include:

- **mpeg_stat**. The original **mpeg_stat** was modified and extended to provide more statistics on the encoded stream. It was also configured to support command-line flags giving the user the ability to control which statistics are generated and how they are output.

- **mpeg_blocks**. **mpeg_blocks** provides a graphical view of statistics relating to the macroblock structure of the encoded stream. **mpeg_play** was extended to include simple video controls and an information window that presents information on macroblock type (intra, forward, backward, bidirectional or skipped), motion vector values and types (full or half pixel), frame types, and average frame sizes.

- **mpeg_bits**. **mpeg_bits** provides a graphical view of bit allocation on a macroblock level. It was built starting with the code for **mpeg_blocks**, and like **mpeg_blocks** is an extension of **mpeg_play** that includes simple video controls and an information window with statistical feedback. **mpeg_bits** provides feedback on individual frame and block sizes and frame-by-frame bitrate. In addition, **mpeg_bits** provides a GUI interface for automatically generating files to modify bit allocation on a macroblock-by-macroblock basis in subsequent encoding.

- CMT Extensions. The Continuous Media Toolkit is a set of objects that allows users to develop networked multimedia applications. Two CMT objects were extended with new functions and options to allow an application developer to extract statistical information about the MPEG-1 video stream being played and to process and present this information to the user.

My own personal contribution to this toolset includes the design and implementation of both mpeg_bits and the CMT extensions.

The remainder of this section presents a more detailed description of **mpeg_stat**, **mpeg_blocks**, and **mpeg_bits**. The CMT extensions are discussed later in a separate section.

2.1 mpeg_stat

mpeg_stat is run with the command line:

```
% mpeg_stat [ -all basename ] [ -start N ] [ -end N ] [ - qscale filename
] [ -size filename ] [ -offsets filename ] [-rate filename ] [ -rate-
length N ] [ -block_info filename ] [-dct] [ -quiet ] [ file_name ]
```

This version of **mpeg_stat**, like its predecessor, parses an encoded stream without actually displaying the frames, collecting and formatting various statistics and printing them to standard output. The updated version provides all of the analysis available from the original and at the same time gathers and presents several new statistics, including:

P and B frames exploit temporal locality exhibited by most video streams. If each frame in a sequence is similar to its neighbors, one would expect the P and B frames to be significantly smaller than the I frames since most macroblocks have a close match in a neighboring reference frame.

Even from this brief description of MPEG-1 it is easy to see that the critical issues for the designers and users of MPEG-1 encoders are:

- What is the best search pattern for reference macroblocks; that is, how do I find the reference macroblock that yields the smallest error term?

- What is the optimal q-scale? How does one hold on to enough data to reproduce a clear image, yet throw enough data away to achieve the needed levels of compression?

- Given an encoded stream, where are the bits going? How large are frames relative to each other? How many non-zero DCT coefficients are in each block? How are individual macroblocks being coded?

A key to answering any of these questions is the ability to analyze the output of the encoder on a very minute level, much deeper than simply checking the size of the entire encoded stream.

1.2.2 Previous MPEG Tools

The MPEG analysis tools discussed in this paper were derived from the Berkeley MPEG-1 Video Player (**mpeg_play**) [8] and an MPEG statistics tool (**mpeg_stat**) [5].

mpeg_play is a software decoder for MPEG-1 video streams. It can handle multiplexed MPEG streams (audio and video together) by ignoring the audio portion.

mpeg_stat decodes the stream and collects various statistics on the encoding, including: total size, IPB frame pattern, total number of frames and frames per second, average frame size, and average bits per pixel for each of the three different types of frame.

While this original version of **mpeg_stat** takes some steps towards providing the kind of feedback discussed above, it does not provide all of the support one might desire. While it gives the total size and average frame size of the stream, it provides no information on how the bitrate fluctuates from second to second, which is a critically important statistic for real-time playback across a network. Furthermore, it does not generate any information on the macroblock level at all. Anyone interested in the fine details of a video sequence encoding, from a compressionist to someone trying to debug an encoder, needs information at the macroblock level, including macroblock size and type, motion vector usage, number of DCT coefficients used, and number of coded blocks.

For a more detailed list of MPEG-1 tools and applications developed by the Plateau Group and other researchers, please refer to the MPEG-1 FAQ [2].

pixels, known as macroblocks, which are then split into blocks devoted exclusively to Y, Cr or Cb.

Each frame is assigned a type, according to a user-defined type pattern. There are three types of frames: I frames, P frames and B frames. In an I (intracoded) frame, each block is converted from a spatial to a frequency domain using the Discrete Cosine Transformation (DCT), concentrating the most significant information at the top left corner of the block. These coefficients are then quantized (i.e. the low order bits are thrown away) according to a user-defined “q-matrix” and “q-scale.” Each coefficient is divided by the product of the corresponding entry in the q-matrix and the q-scale, and the result is rounded down to the nearest integer. This is a lossy step in the compression, and gives the user a mechanism for trading quality for bitrate by modifying the q-scale. The use of DCT and quantization together exploits spatial locality: when colors do not change much from pixel to pixel, the frequency is smoother and more regular across the block. It can therefore be represented with relatively few high-order DCT coefficients.

In a P (predicted) frame, where possible, macroblocks are encoded relative to a previous “reference” frame, which must be another I or P frame. The encoder, using a search algorithm, looks for macroblocks in the reference frame that are similar to the macroblock in question. If one is found, the block is encoded as a motion vector (i.e. the offset from the macroblock being encoded to the reference macroblock) and an error term. The motion vector may be specified in units of whole or half pixels. If a suitable reference macroblock cannot be found, the macroblock is encoded as in an I-frame.

Finally, a B (bidirectional) frame is encoded relative to a previous reference frame and/or the future reference frame. The previous reference frame is the last I or P frame before the B frame, and the future reference frame is the next I or P frame after the B frame. The encoding of macroblocks in B frames is similar to the encoding in P frames, except that the macroblock might be represented by two motion vectors (i.e. the macroblocks are averaged), a vector to a future macroblock, a vector to a previous macroblock, or a straight DCT coding.

Figure 1 indicates how a series of I, P and B frames would be encoded relative to eachother.

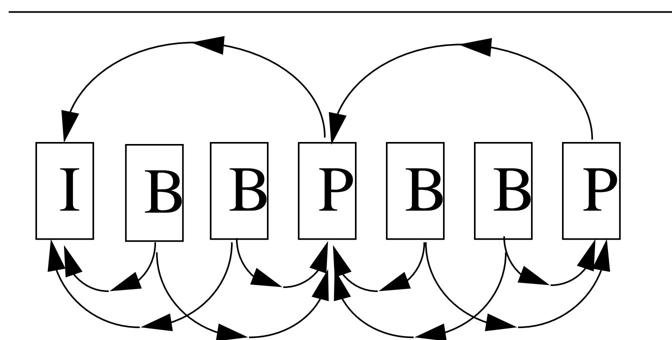

Figure 1: frame dependencies

These questions are not trivial. For example, consider an application that requests encoded video from a server, then decodes and plays the video at a remote client. Suppose further that the client-server connection has a fixed bandwidth, as in a typical interactive television system; it is easy to see how an unexpected series of abnormally large frames could prevent the application from meeting its frames-per-second demands.

In addition, the compressionist should be able to develop a feel for how the compression algorithm in question behaves for different types of inputs. If he can somehow see the internals of a compressed stream - exactly how the bits were allocated, what choices were made in how to take advantage of redundancy, etc. - he can gain an intuition about how well any given video clip might compress: what kind of images or motion tend to be troublesome, for example. Furthermore, compression standards allow for different user-defined parameters that modify the encoding algorithm in a number of ways. If the compressionist can get detailed feedback on the structure of an encoded stream, he can better understand the effects of varying these parameters and therefore be better equipped to fully exploit them.

The Plateau Multimedia Group at UC Berkeley has developed a series of tools to meet these needs, enabling a compressionist to examine interactively the structure of video streams encoded according to the MPEG-1 standard [3]. This paper describes the design and implementation of these tools, as well as future directions in this area.

The remainder of the paper is structured as follows. Section 1.2 provides an introduction to the MPEG-1 standard and a set of MPEG applications that provided the building blocks for the tools under discussion. Section 1.3 describes the tools we developed. Section 2 gives more detailed examples of the analysis tools, and Section 3 discusses implementation issues for these tools. Section 4 describes extensions to the Continuous Media Toolkit [7] that allow users to build their own analysis tools. Finally, Section 5 provides some further discussion and directions for future work.

1.2 Background

1.2.1 MPEG-1 Standard

The MPEG-1 video compression standard is designed to provide VHS quality color video playback. MPEG-1 exploits both the spatial redundancy within each frame of a video sequence (the likelihood that neighboring pixels will be similar) and the temporal redundancy between successive frames (the likelihood that neighboring frames will be similar). The following discussion briefly describes how these redundancies are exploited, enough to give some insight as to the statistical feedback in which a compressionist might be interested. For a more complete discussion of MPEG-1 encoding, refer to [1].

The input for an MPEG-1 encoder is typically an ordered sequence of frames, each frame represented by a series of 24 bit RGB values, one for each pixel in the frame. The RGB values are first converted to 12 bit YCrCb format, allocating 8 bits for luminance and 2 bits each for Cr and Cb information. Each frame is then broken into 16x16 blocks of

Analysis Tools for MPEG-1 Video Streams

Doug Banks and Lawrence A. Rowe
Department of Computer Science - EECS
University of California at Berkeley
Berkeley, CA 94720-1776
(dbanks@cs.berkeley.edu, larry@cs.berkeley.edu)

Absract

Several tools are described that analyze the encoding of video streams compressed according to the MPEG-1 standard. The relative strengths and weaknesses of each tool are discussed, along with the problems solved in implementing them. Finally, several statistics-gathering extensions to the Continuous Media Toolkit (CMT) are introduced. These extensions provide a powerful, highly configurable, and conceptually simple framework for the design of customized MPEG-1 analysis tools.

1.0 Introduction

1.1 Motivation

The rise of multimedia applications capable of presenting video streams has given rise to a new job description: the “compressionist.” Since video streams tend to be very large, but they typically exhibit a good deal of redundancy (since any given pixel is likely to be similar to those pixels surrounding it, and any given frame is likely to be similar to those frames surrounding it), it is standard procedure to compress the video used in an application. The compressionist is the individual responsible for processing raw video into compressed video, a form usable by the target application.

The compressionist has two competing objectives: quality and size. The compressed stream should look as good as possible when decompressed; optimally, it should be indistinguishable from the raw video. On the other hand, the encoded stream should be as small as possible. The size of the encoded stream is important if the multimedia application must fit within a certain fixed space, for example because it will be shipped on a CD-ROM. Even in an environment where storage constraints are not strict, for example a video server, encoded streams need to hold consistently to a constant bitrate, given the real-time demands of video playback and the limited bandwidth to clients

For a compressionist to check the quality of a compressed video is fairly simple. Checking size and bitrate constraints are more complex issues, however. The total size of an encoded stream is certainly readily available, but that gives no insight into the internals of the encoding. How big is each frame? Are several abnormally large frames offset elsewhere by abnormally small frames? What regions in a given frame compress the worst, i.e. require a relatively large number of bits to encode?