EFFICIENT BUFFERING CONTROL FOR A SOFTWARE-ONLY, HIGH-LEVEL, HIGH-PROFILE, MPEG-2 DECODER

By

YISHU HE

A THESIS PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2002
This is dedicated to my parents.
ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Jonathan C Liu and Ju Wang for his constant encouragement and the opportunity to work on this research project. His innovative ideas and encouragement have made this work interesting and challenging. I would also like to thank Dr. Randy Chow and Dr. Jih-Kwon Peir for agreeing to serve on my committee. I thank my parents for their inspiration and support throughout my academic career. I sincerely thank all the people who have helped and supported me directly and indirectly in the course of this thesis work.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>vi</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Define Scalability</td>
<td>1</td>
</tr>
<tr>
<td>1.2 State of the Art</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Experimental Result</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Revised Buffer Scheme</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Organization of the Paper</td>
<td>6</td>
</tr>
<tr>
<td>2 RELATED STUDY</td>
<td>7</td>
</tr>
<tr>
<td>3 PROBLEM NATURE AND ANALYSIS</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Memory Usage Analysis</td>
<td>14</td>
</tr>
<tr>
<td>3.2 Impact of Memory Shortage</td>
<td>16</td>
</tr>
<tr>
<td>4 EFFICIENT BUFFERING SCHEMES</td>
<td>21</td>
</tr>
<tr>
<td>4.1 The First Solution</td>
<td>21</td>
</tr>
<tr>
<td>4.2 Implementation and Experiment Result</td>
<td>24</td>
</tr>
<tr>
<td>4.3 Further Optimization in the Slave Nodes</td>
<td>29</td>
</tr>
<tr>
<td>5 CONCLUSION</td>
<td>34</td>
</tr>
<tr>
<td>APPENDIX</td>
<td>35</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>44</td>
</tr>
<tr>
<td>BIOGRAPHICAL SKETCH</td>
<td>46</td>
</tr>
</tbody>
</table>
There are some common video resolutions available today. Typical ones include QCIF (352*240), CIF (704*480), (1024*1024) and (1408*960). We believe that a high-quality MPEG-2 software decoder should support a good scalability performance across different video resolutions. By using our parallel software-only MPEG-2 decoder, scale-down performance has been proved effective for low-level and main-level MPEG-2 streaming videos. However, the challenge arises when attempts are made to support (1024*1024) and (1404*960) high-level MPEG-2 video. It is found that the existing scheme suffers significant performance degradation when decoding high-level MPEG-2 video with full system configuration. The origin of the problem is traced to the excessive memory usage of the original design of the parallel scheme. Therefore we propose an efficient buffer management mechanism such that the memory requirement can be reduced by 50%. This is approached by two steps: first we use an ST scheme to minimize the transmission buffer in a slave node by allowing dynamic sharing between frames in one GOP; then we further reduce the buffer space by a dynamic on-demand
allocation on the slave side. By solving the memory-shortage bottleneck, we have proven that scale-up performance can be successfully achieved with 13 and 14 slave nodes for the high-resolution (1024*1024) and (1404*960) video formats.
CHAPTER 1
INTRODUCTION

1.1 Define Scalability

Many streaming video formats are used intensively in today’s society. Commercial streaming formats such as RealPlayer and Windows Media Player are common tools to display low-resolution (e.g., 352*240 without dithering) video over Internet. However, due to the rapid deployment of high-speed networks (e.g., ATM networks) and “first-mile” technology (e.g., cable modem and digital subscriber lines), users have the capability for receiving high-quality and high-resolution streaming videos to their desktop or TV sets.

Beyond Realplayer and Media Player, the great success of DVD titles bring us the widely-accepted MPEG-2 video formats. According to MPEG-2 specifications, a wide range of video resolution is possible. In reality there are usually five formats that are widely used: 170 * 120, QCIF (352*240), CIF (704*480), 1024*1024 and 1408*960. The first three formats represent the most dominant applications: video conferencing, low quality streaming video and broadcast level video (DVD). The latter two video formats are projected to be used for future HDTV and high-end video applications.

Therefore, in the near future, we envision that a high-quality MPEG-2 software decoder needs to support good scalability performance across different video resolutions. For example, video resolutions should be supported from medium resolutions (e.g., 704*480) to large resolutions (e.g., 1404*960) with guaranteed display quality. Smooth display rates with more than 24 frame per second (fps) should be guaranteed. Ideally, an MPEG-2 software decoder should also automatically choose the best resolution/quality to adapt to the user’s
environment settings. For instance, if machines are equipped with sufficient CPU, we should deliver as high a resolution as possible.

MPEG-2 specifications do provide some recommendations for "scalable" coding where video can be reconstructed on the basis of user demand to suit different application scenarios, e.g., for different available communication bandwidths. This is implemented by multi-layer encoding/decoding with the assumption that the video resolution size remains the same. However, the enlargement of the video resolutions usually generate a need to have more refined quality associated with the frame size. Therefore, MPEG-2 defined a family of video formats using a profile/level combination.

The concept of profile in MPEG-2 can be roughly interpreted in the view of precision of pixel, and the level corresponds to spatial resolution. In addition to the spatial dimension, video quality can be improved by representing the pixel more precisely. This can be accomplished by allocating more bits for each pixel, i.e., using more color and/or more precise quantization. In this paper, we mainly target for high-profile high-level MPEG-2 formats. The recommendations of MPEG-2 scalable features can potentially increase the decoding complexity. Among these, increasing the size of video frame resolution seems to have the most direct influence on the decoding performance because more macro-blocks need to be decoded. We believe the increase in spatial resolution is probably the most effective way to support better video quality. Thus, as part of the long-term investigation, the goal of this paper is to examine how spatial scalability can be supported with different video resolutions.

1.2 State of the Art

We have been researching a generic, portable, pure-software MPEG-2 encoder/decoder for the last few years. We believe a pure-software based MPEG-2
decoding is still desirable in many situations for its flexibility and scalability. However, software only MPEG-2 decoding is very computation intensive, especially for high-level video format. For example, a high-profile (1440*1152) base MPEG-2 video contain 4 times as many macro blocks than the main level DVD video, roughly corresponding to 4 times more decoding computation. With an enhancement layer of the same spatial resolution (SNR scalability), the complexity of the decoding process will be doubled. Thus we expect an 8 fold increase in computation requirements for such video formats. This computing gap will not be covered in the near future according to the current microprocessor evolution trend.

To achieve high performance software MPEG-2 decoding, we had designed a parallel MPEG-2 decoder that can run on both cluster and multi-processor environments [1, 2]. With a high-speed network, a parallel decoder could produce high decoding frame rates by distributing the decoding workload into several computing nodes. The pipeline scheme [1] takes a Master/Slave architecture where the master is in charge of data distribution/collection and the slave nodes perform MPEG-2 decompression algorithms for the assigned task. The master also maintains the smooth running of the pipeline to assure the highest overall system throughput.

1.3 Experimental Result

The results were very promising with 30-fps playback achieved with 4 Pentium 400MHz desktop computers, and a 72-fps HDTV frame rate achieved in a SUN SMP environment. However, only one video resolution was investigated with a main level MPEG-2 format (i.e., 704*480). It remained unclear how our parallel MPEG-2 decoder would support larger (or smaller) video resolutions. Did we expect the same software to be used for (1404*960) without any adaption? How many slave nodes are required to deliver a 24-fps high-level high-profile MPEG-2 video with (1404*960) resolution?
By producing several versions of the same video content, we are able to generate different video resolutions from (352*240) to (1404*960). The first two resolutions are (352*240) and (704*480), which roughly corresponding to low-level QCIF and main-level CIF formats of MPEG-2 standards. Our parallel MPEG-2 decoder performs well on these streaming videos. Three representative video content with different characteristics are decompressed up to more-than-200-fps for (352*240) and 75 fps for (704*480) using 14 slave nodes. Therefore, the performance results indicate that our parallel MPEG-2 decoder does scale-down well for low-level and main-level MPEG-2 streaming videos.

The challenge arises when attempts are made to support (1024*1024) and (1404*960) high-level MPEG-2 video. We have observed a severe performance degradation (e.g., dropping from 18 or 20 fps to 2.5 fps) when more than 10 slave nodes are used. It is not trivial to us why this behavior happens, and we are perhaps one of the first groups that discover this strange system behavior. By analyzing the runtime system resources utilization, we found that the system memory is quickly exhausted when increasing the number of slave nodes. When decoding the video file with high spatial resolution, the increase of memory usage eventually becomes a system bottleneck.

We observed that at the saturating state, the operating system spends most of its CPU time swapping in/out between main memory and secondary storage. The analysis of the original data pipeline scheme also indicated that the problem will become more severe when a larger GOP size is used with large video frame sizes. Therefore, in addition to the already-found network bottleneck from Wang and Liu[1], we discovered that lack-of-memory can also be another system bottleneck.

### 1.4 Revised Buffer Scheme

To address the challenge and obtain high scalable decoding for high resolution video, we proposed and implemented two revised memory management approaches
to reduce the buffer requirement. The first is Minimum Transmission Buffer in Slave Node (ST scheme). In our original design, the slave nodes allocate the buffer for the whole GOP. When the number of nodes grows, a lot of memory is needed. To reduce the memory requirement, we reduce the transmission buffer size of the slave nodes to three frames. We can see the benefit of the ST scheme from the decreased page faults of the slave nodes and the increased decoding frame rate.

In the ST scheme, we use a 3-frame transmission buffer for each slave node. For scalable MPEG-2 streams, each L-layer sub-stream requires the same amount of buffer space as that of the base layer. It can be expected that memory will become a bottleneck again. To further reduce the buffer requirement in slave nodes, we proposed a dynamic buffer requirement. It is obvious that only the B-frame needs the whole three frame buffer. So if we allocate the buffers according to the actual picture need, the effective number of frames per buffer will be only 85% of the 3 frame buffer. Furthermore, dynamic buffer allocation can be applied inside the decoding of each frame. The experimental results show that the buffer space is significantly reduced, and we observed a well-scaled decoding performance for the high-resolution MPEG-2 video.

With the revised buffer schemes, our parallel decoder is able to deliver high quality scalable decoding performance based on the configuration of slave nodes. In order to achieve the 24-fps target decoding frame rate, we need 2 slave nodes for (352*240) video resolution, 5 slave nodes for (704*480) main resolution, and 13 and 14 slave nodes for the high-resolution (1024*1024) and (1404*960) video formats respectively. We also observed that the system resource usage at large scale settings is under control, indicating the system can be easily scaled up, as well as scaled down.
1.5 Organization of the Paper

The organization of the paper is as follows: section 2 provides related studies and a brief overview of MPEG-2 scalability. Section 3 describes the preliminary results of scalable decoding performance for various MPEG-2 video formats. The nature of the problem is identified by analysing the original buffer scheme and runtime system statistics (CPU usage, memory occupation). In section 4, we present the two improved buffer schemes and report the experimental results. Finally, section 5 gives the conclusion of this paper.
CHAPTER 2
RELATED STUDY

The optimization of MPEG-2 decoding has been attempted in both software and hardware approaches. Based on general purpose microprocessor, much of the work have been focused on accelerating huffman decoding, fast IDCT, and other run time cost, such as the work in Lee [3]. In Soderquist and Leeser [4], the memory access pattern of MPEG-2 decoding was analyzed to improve the cache efficiency, their proposed cache-oriented architecture reported to reduce memory traffic by 50%. However, the real-time performance requirements was not addressed in their work. In Patel [5], performance of a software decoder was discussed and various enhancements in IDCT, ME and DITHERING were studied. However, only a (320*240) video stream was decoded in real-time.

Beside the pure software-oriented optimization, many CPU vender had built multimedia instructions inside the general purpose processor [6, 7, 8]. Lee [3] reported a 4 folds performance improvement using the PA-RISC multimedia instructions. Recently, INTEL’s MMX technology is gaining more interests in MPEG-2 decoding optimization [9, 10, 11]. In our experiments, a 70% reduction of execution time is observed for IDCT transform. With the Pentium III 700MHz CPU, the main level MPEG-2 video (DVD quality) can be decoded at nearly jitter-free quality.

Some commercial software DVD decoders can operate on a lower CPU clock rate with hardware multimedia support features provided by a video card. For example, most state-of-art video card vendors had integrated IDCT and even motion compensation into their chips [12]. These hardware features can significantly relieve the computation load to the host CPU.
Pure-hardware approaches usually use a redundant DSP unit and a much wider internal bus design, which make it possible to exploit instruction-level parallelism (such as VLIW). Some of the works are reported in Akiyama and Sriram [13], and Sriram and Hung [14]. In Baum et al. [15], a low-cost, high-performance RISC processor core based chip set is proposed to encapsulating many of the functions required in high quality consumer audio-visual platforms. In Ishiwata et al. [16], A single-chip MPEG-2 MP@ML codes, integrating 3.8M gates on 72mm is described. It has heterogeneous multiprocessor architecture in which six microprocessors with the same instruction set but different customization execute specific tasks such as video, audio etc. concurrently. The microprocessor, developed for digital media processing, provides various extensions such as VLIW one and DSP one inherent in its architecture. Making full use of the extensions, the chip executes encoding and decoding of video, audio and system concurrently in real time.

However these approaches did not address high quality scalable MPEG-2 video formats which will probably become more desirable in the future multimedia applications. Moreover, their strong dependence on specific hardware make them less flexible and reusable. In some cases, a generic pure software solution is more desirable. As demonstrated in the literature, pure software MPEG-2 encoding/decoding requires large amounts of computation power. Much has been done [17, 18, 19] to parallelize the MPEG-2 encoding process based on SMP environment or clusters of workstations. With Intel’s Paragon multiple processor system, Akramullah et al. [18] reported a real-time parallel encoder for low resolution MPEG-2 encoder. Gong and Rowe [19] proposed a coarse-grained parallel version of a MPEG-1 encoder and showed a very good parallel gain. In He et al. [20], the schedule algorithms of parallel MPEG-4 encoding were discussed to
balance the system load when dispatching multiple video streams over a cluster of
workstations.

On other hand, Only a few works have been reported regarding parallel
MPEG-2 decoding. A parallel MPEG-2 decoder based on a shared-memory SMP
machine was reported in Bilas et al. [21], however, they did not address how real-
time decoding could be supported, and whether the system can be scaled up for the
high-profile and high-level video source. In Wang and Liu [1], we proposed a data
pipeline based scheme towarding pure software, scalable MPEG-2 decoder. The
early results show that MP@ML MPEG-2 video can be adequately supported with
low end CPUs.

However, the decoding performance of high end MPEG-2 video formats
with scalable features has not been reported. The high end MPEG-2 video
format usually comes with multiple substreams, with a mandatory base layer
and additional enhancement layers providing various scalability features. Three
scalabilities are defined so far: SNR, Spatial, and Temporal scalability. To enable
these features, additional computation resources must be provided. Therefore it is
not clear to us that high-profile high-level could be automatically supported with
the existing solutions. It is our goal in this study to verify how scalable decoding of
high resolution video can be achieved.
CHAPTER 3
PROBLEM NATURE AND ANALYSIS

We have long suspected that the scalability issue for high-profile high-level MPEG-2 video could be a challenging issue, but it was not until recently that we found this practical issue does exist. By using a public domain MPEG-2 encoder, we were able to generate a series of MPEG-2 video streams with different resolutions. The video content was encoded with $N=12$, and $M=3$ with chroma format $4:2:0$. Each video content had seven versions with different resolutions, from $352 \times 240$ to $1404 \times 960$. The intermediate resolution was chosen so that continuous performance trends could be observed. We used the same GOP structure, quantization table, color format, and motion search range as the encoding parameters to have a fair comparison. Each encoded video consisted of sixty frames, which is roughly four GOP.

The tested video sources consisting of three different contents of different motion activity and picture complexity were chosen. The "flower" video type consists of a static scenario of flowers. The "calendar" title has slow motion and a complex picture. The "tennis" is the most motion intensive one. Each of the three video titles was encoded into the four different sizes we are interested in. All the encoding parameters were the same except the horizontal and vertical sizes.

Using the performance model in Wang and Liu [1], we can derive the expected decoding performance. To simplify the discussion, we assume a one layer structured MPEG-2 video file. With the assumption of sufficient long video sequences, the expected decoding frame rate can be approximated by the following:

$$ FRD = \frac{(N * D)}{\max\{D.T_{\text{single}} + T_{ms} + T_{sm} + 2c, N.(T_{sm} + T_{ms} + 2c)\}} $$
where $N$, $D$ denote the engaged processor number and the length of GOP (Group of Picture). $T_{single}$ is the average decoding time of one frame of the given video file at a given CPU. $T_{sm}$ and $T_{ms}$ are the transmission time of a decompressed frame and raw frame respectively (equation (7), (8) in Wang and Liu [1]). Using the same hardware configuration as in the SUN SMP environment in Wang and Liu [1], the expected decoding performance is shown in Table 3.1.

Table 3.1: Expected Parallel Decoding Performance

<table>
<thead>
<tr>
<th>Video Spatial Resolution</th>
<th>2 node</th>
<th>4 node</th>
<th>8 node</th>
<th>16 node</th>
</tr>
</thead>
<tbody>
<tr>
<td>352*240</td>
<td>25 fps</td>
<td>55 fps</td>
<td>120 fps</td>
<td>260 fps</td>
</tr>
<tr>
<td>704*480</td>
<td>10 fps</td>
<td>25 fps</td>
<td>48 fps</td>
<td>92 fps</td>
</tr>
<tr>
<td>1024*1024</td>
<td>3 fps</td>
<td>8 fps</td>
<td>20 fps</td>
<td>34 fps</td>
</tr>
<tr>
<td>1404*960</td>
<td>3 fps</td>
<td>6 fps</td>
<td>14 fps</td>
<td>25 fps</td>
</tr>
</tbody>
</table>

Though the expected decoding performance can be predicted via our proposed performance model, we are interested in whether the experimental results will agree with us. By using a SUN SMP machine with 14 248-MHz UltraSparc CPUs, 512-MByte memory space and internal communication bandwidth up to 680 Mbps, we have collected the scalability performance with different video resolutions.

The achieved frame rates for the low- and main- level MPEG-2 video are very close to our prediction. The results showed only a slight difference among the three video contents.

For the small resolution video (Figure 3.1.a), we observed a linear increasing of frame rate. The maximum frame rate is achieved when 14 nodes are deployed, providing 220 fps for *tennis*, 231 fps for *calendar*, and 213 fps for *flower*. Since the video size is small, the system’s theoretical peak could reach 500 fps (at 30 nodes) according to the prediction in Wang and Liu [1]. Our test platform only has 14 nodes, thus the theoretical saturation point will not be reached. The results for the main-level video (720*480) also conform with our prediction. The performance for the three video titles shows little difference in terms of frame rate. Each of them
Figure 3.1: Decoding Performance For Low Resolution Video (a) Decoding performance for 352 * 240 video (b) Decoding performance for 704 * 480 video

increases close to linearly when more slave nodes are used. The highest frame rate achieved is 70 fps (with 14 nodes).

However, the scalability performance for the high resolution MPEG-2 videos are not satisfactory. In Figure 3.2.b, the decoding rates for (1404*960) MPEG-2 files are illustrated. Starting with 2 fps at single node configuration, a linear increase can be observed. The highest decompression rate is 20 fps for "flower" at 9 slave nodes, and 22 fps for "tennis" and "calendar" at 11 nodes. With 10 slave nodes, the decoding performance of "flower" suddenly dropped to 2.5 fps, and continued deteriorating with a small rebound at 11 slave nodes. For "tennis" and "calendar", a similar performance degradation is observed at 12 slave nodes, right after the peak performance.
Figure 3.2: Decoding Frame Rate For High Resolution Video (a) 1024 x 1024 (b) 1404 x 960

Note that 20- or 22-fps decoding performance is not considered as a real-time video display. It is considered closer to slow motion, which can not be synchronized with an accompanying audio track. Therefore, it is desirable to achieve at least 24 fps, which is close to theater’s film display. Most importantly, the display rates should be smooth and without sudden drops as illustrated by the 2.5 fps that we observed at this time.

The behavior is also observed with other high-profile formats. Similar performance drops are observed for the (1024*1024) format. The system can do well for up to 10 slave nodes, where a peak of 23 fps can be achieved with 10 nodes. However, great degradation occurred after 11 nodes. The frame rate dramatically drops to only 2 fps, which is even worse than a single slave node configuration.
Further increasing the slave node seems not to improve the performance at all. We observed no frame rate improvement after 11 nodes.

3.1 Memory Usage Analysis

The performance degradation shows our pipeline scheme is bounded by a system bottleneck. We need to find the underlying reason for this to further improve the original design, the goal here is to make the system well scale up for the high-level high-profile MPEG-2 video. According to Wang and Liu [1], the data exchange between the master node and slave nodes is based on GOP, which usually contains 10 to 20 frames. The slave node has to keep a buffer space to accommodate both the compressed and decompressed video frames, for each of the GOP. In the master node, a dedicated buffer space (1 GOP) is reserved for displaying, and another for receiving data from slave nodes. The following Figure 3.3 depicts the buffering requirements and relations between the master and slave nodes:

The memory requirement for the master node is:

\[ M_m = m_c + m_{\text{streambuffer}} + m_{\text{outbuffer}} + m_{\text{inbuffer}} \]

Here \( m_c \) is the size of executable code for the master, about 500 KB. \( m_{\text{streambuffer}} \) is the streaming buffer to receive the compressed video packet from the video server, we currently fixed it to be 1 MB. \( m_{\text{outbuffer}} \) and \( m_{\text{inbuffer}} \) are dedicated for information exchange in the parallel decoding. \( m_{\text{outbuffer}} \) equals one GOP of MPEG-2 compressed frames, and \( m_{\text{inbuffer}} \) needs to accommodate two GOP of decompressed frames (one GOP for displaying and another for incoming traffic). Notice that in our scheme, the outbuffer and inbuffer are shared by all slave nodes, which is made possible by the master doing a round robin polling. Using the horizontal video size \( h \) and vertical video size \( v \), GOP=15, and the average
Figure 3.3: Memory Usage Illustration

compression ratio $\lambda$ is 20, we have:

$$M_m = 0.5 + 1 + \lambda * h * v / 10^6 + 2 * GOP * h * v / 10^6 (MB)$$

$$= 1.5 + (\lambda + 2 * GOP) * h * v / 10^6 (MB)$$

For the slave processes, the size of executable code is also 500 KB. The stream-buffer is not used since the slave node does not receive compressed video packets from the video server. A compressed data buffer is used to receive data from the master (the same size as the $m_{outbuffer}$ in master node). The transmission buffer can serve two purposes, it is used during the decoding processing, thus obviating the need for separate space for the YUV components of each macroblocks, and so
that in-place transmission can be done without moving data. Using a 4:2:0 color scheme, the average bits per pixel is 12 bits, instead of 8 bits used in the display system, thus the transmission buffer $m_t$ is 1.5 times the size of $m_{inbuffer}$. We have

$$M_s = m_c + m_{compressedbuffer} + m_{transmissionbuffer} = m_c + m_{outbuffer} + 1.5 * m_{inbuffer}$$

Using the above two equations, we can calculate the memory requirement for the master node and the slave node for each video format. For the test stream tennis40 (1404*960), $M_m = 42$ (MB), and each slave needs about 30.8 MB. The original consideration of pipeline parallel design is to minimize the communication cost and reduce the number of high level network access times. However the memory buffering scheme is not considered optimized.

The accumulative buffering space will grow quickly when using a large scale slave node configuration, which causes unsatisfactory scalability performance when the number of slave nodes is large. For instance, let $N$ be the number of slave nodes, the total memory requirement becomes

$$M_t = M_m + N * M_s$$

Using the parameters of our testing MPEG-2 video, the actual memory used is listed in table 3.2. In Figure 3.2.b and 3.2.a, we find that the tennis40 has the frame rate dropped when $N=9$, and tennis60 dropped at $N=11$. The corresponding amount of memory used is 319.9 MB and 296.5 MB respectively. The minimum of these two should be used as the indication of potential memory outrage. This amount of memory is actually 70% of the system physical memory.

### 3.2 Impact of Memory Shortage

The performance impact of a non-optimized buffering scheme will affect on the competition between user processes (e.g., our communication and decompression software) and system processes (e.g., demand-paging mechanisms by OS). Because
Table 3.2: Memory requirements for different nodes (MB)

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Horizontal resolution</th>
<th>Vertical resolution</th>
<th>1 node</th>
<th>2 nodes</th>
<th>4 nodes</th>
<th>8 nodes</th>
<th>9 nodes</th>
<th>10 nodes</th>
<th>11 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tennis40</td>
<td>1404</td>
<td>960</td>
<td>73.5</td>
<td>104.3</td>
<td>165.9</td>
<td>289.1</td>
<td><strong>319.9</strong></td>
<td>350.7</td>
<td>381.5</td>
</tr>
<tr>
<td>Tennis60</td>
<td>1024</td>
<td>1024</td>
<td>59.5</td>
<td>83.2</td>
<td>130.6</td>
<td>225.4</td>
<td>249.1</td>
<td>272.8</td>
<td><strong>296.5</strong></td>
</tr>
</tbody>
</table>

the shortage of the overall memory, the system process will generate a significant number of page faults, which in general slowing down the decompression speed due to the lack of CPU. The shortage of system memory will force the operating system to swap some of the memory page out to hard disk, and this activity in turn will use more CPU time, thus affecting the performance of all user space processes.

The evidence from system runtime statistics can be collected from the CPU time distribution and the number of page faults to support this unique observation. Figure 3.4 illustrates our measured number of page faults versus the number of slave nodes. For the sake of clarity, we only present the results for “tennis”, the “flower” and “calendar” shows similar results for this measurements. We plotted the page faults for the four video resolutions, from QCIF to MPEG-2 high level (1404*960).

The following observation can be made:

- For the 352x240 video, the page faults virtually remain unchanged, and are kept at a low level (1010 page faults/frame). Increasing the video resolution to 704x480 is reflected by the rise in the number of page fault, a four fold jump is observed. Nevertheless, the 704x480 case still has a flat curve for the increasing slave node, indicating the system is running steadily.

- For the 1024x1024 video, the number of page faults increased considerably at beginning, but still within a manageable level. 1200 page faults per frame is observed for 2 slave nodes, and remain the same until 9 slave nodes. This is followed by a significant increase at 10 to 12 slave nodes, reaching 3500 page faults per frame at 12 slave nodes as peak. Then the figure drops back to certain degree, but still maintains a high level (more than 2500). Compared with the decoding performance in Figure 3.2.b, the period of high page faults coincides with the collapse of the decoding rate. This indicates that the excessive page faults had driven the system into an outrage state.
The page faults behavior of the 1404x960 video shows the same pattern as in the 1024x1024 case. The former one does have a the higher number of page faults than other cases, due to its highest spatial resolution thus high memory requirements. The outrage of the page faults occurred even earlier than the 1024x1024 case, with the jump between 9 and 10 slave nodes. The most frequent page faults reached 4700 faults/frame at 10 slave nodes, where the decoding performance drops from 22 fps to 2.5 fps (see Figure 3.2.a).

Figure 3.5 presented the overall CPU usage distribution between user space process (our decoding algorithm), system cost (paging), and system idle time. With one slave node, 90% of the system time is idle, 8% of the CPU time is used in the user space, and the remaining 2% for other system maintenance. With increasing slave nodes, the user space time increases proportionally, and the system idle time decreases. During these periods, more CPU time is used for the slave nodes, and the decoding frame rate increases linearly. After 8 slave nodes, however, both system idle time and user space time dropped significantly, while...
the system overhead showed a sharp rise. About 90% of the CPU time is used by the operating system, while user space only occupies 5% of CPU time. Recalling that the page faults number increases suddenly at 9 slave nodes (see Figure 3.4), we conclude that the system spend most of its CPU time swapping page in/out, thus the observed drop of decoding performance.

The previous sections show that our original parallel decoder could consume a huge amount of memory space as a frame buffer when decoding high-resolution MPEG-2 video. The always-limited physical memory can be exhausted when a large number of slave nodes are deployed. However, in order to reduce the communication overhead, a large GOP will make the memory over-allocation even worse. The system page fault and CPU time distribution provides strong evidence for this claim. Thus, how to create efficient buffering schemes becomes critical.
to the success of the *data pipeline scheme* for supporting high-profile high-level MPEG-2 video decompression.
In order to propose efficient buffering schemes, we have to analyze more deeply our original scheme. In our original design [1], each slave node allocated enough buffer spaces at the initialization stage. The buffer is big enough to hold a GOP of frames. These buffer space will be allocated statically throughout the life time of the video decompression of current GOP. When the decoding of the whole GOP is completed, the data in the buffer (decompressed video frames) will be sent into the MPI communication protocol stack.

In the master node, a two-buffer scheme is adopted, where one buffer is used for incoming frames from slave node, another buffer is dedicated for displaying the last GOP of frames. To avoid the memory movement (which is undesirable for video display), we actually swap the incoming frame buffer and display buffer every time a new GOP of frames is received.

Therefore, the memory requirements of each slave node depends on the GOP length, and the picture size. The aggregate memory for the whole system will increase linearly when the number of nodes grows.

4.1 The First Solution

To reduce the memory requirements in slave nodes, the buffering scheme should be redesigned. Ideally, a perfect buffering scheme might only need one transmission buffer. However, due to the decoding dependency inside the MPEG-2 video structure, we are not able to use only one frame buffer in reality. To decode a B-frame, we need at least two reference frames, this indicates that the worst case of the minimum buffer should be three frames, with two frames for reference frames,
and one for the working B-frame. With a careful redesign of the master-slave communication protocol, using a 3-frame transmission buffer in the slave side is possible, which we called the ST scheme. When the picture size is 1024*1024 and GOP=15, we can save about 12 MB buffer space per slave node, about an 80% reduction in the slave side.

To adopt the proposed memory-efficient algorithm, the slave decoding process needs to rotate the usage of two reference buffers as suggested in the reference serial decoder. Let forwardb pointing to the forward predicting frame, and backwardb to the backward predicting frame from the view of B-Frame decoding, the following rotation rules must be obeyed:

1. The first frame, I-Frame, is decompressed to the forwardb, which is initialized pointing to the first buffer.
2. The first P-Frame will refer to forwardb, and is decompressed to backwardb, initialized pointing to the second buffer.
3. The following P-Frames will use backwardb as reference frame, and are decoded into the forwardb. After completion, forwardb and backwardb have to be switched.
4. For each B-Frame, it will refer to forwardb as forward predicting and backwardb as backward predicting frame. The decompressed data is stored in the third buffer.

Also the slave has to send a frame back to the master once it is completely decoded, so that the buffer can be reused for the next frame. As to the master side, the master should be able to receive the data whenever it appears in the network layer, such that the slave nodes don’t have to wait on the blocking transmission. This can be implemented via the use of UNIX SIGNAL mechanism.

The minimum required frame buffer can be further decreased from 3-frames to 2 frames. For the I- or P- frames, we need one buffer for the prediction picture, and another buffer for the working frame. The two buffers change their role after
decoding a I- or P- frame, so that the most recently decoded I- or P- frame is used as the prediction frame for the next P- frame. For the B- type frame, since the result is not used as reference, we can directly send each decoded block into the MPI sending protocol. The above discussion assumes the reference frame of P-frame is always the last decoded P- frame, and the reference frame for B- frame are the last two P- frame. Nevertheless, this approach also works if the B- and P-frames always refer to the I- frame with the proper setting of the reference buffer.

With this scheme, the expected memory requirement become

\[
M'_t = M'_m + N \times M'_c
\]

\[
= M_m + N \times (m_c + 1.5 \times 3 \times m_{frame} + m_{inbuffer})
\]

Here the buffer space of the master node remains the same. For different video content, the expected amount of \(M'_t\) is plotted in Figure 4.1.

![Figure 4.1: Memory Size Using Minimum Transmission Buffer](image)

It is observed that the memory required increases at a slow slope, where each slave node will introduce only about 6 MB additional space for the tennis40 video. The tennis60 finds an even smaller memory requirement. To find out the number of maximum slave nodes before system memory runs out for the 1404*960 single
layered MPEG-2 video, we have \((47 + (N-1) \times 6) < 300\) MB (use 300 MB as a system threshold). This will give \(N=43\) slave nodes and more than 60 fps..

### 4.2 Implementation and Experiment Result

We implemented the \(ST\) memory allocation scheme, and repeated the experiments for the high level MPEG-2 video with the new buffer management. The measured actual memory requirement is depicted in Table 4.1. The percentage of the new memory requirement and the original memory size is given below the actual memory size. The memory requirement for the \(ST\) is significantly less than the original scheme, especially when more slave nodes are deployed. For 1 slave node, we need 53.5 MB for 1404*960 video, which is 27% less than the original one. For 4-slave-node case, the \(ST\) scheme use 85 MB instead of the original 165 MB, which is almost 50% memory saving. This closely matched the analytical estimation in the previous section, since the \(ST\) scheme can save 66% memory in slave nodes. The overall saving will always be below 66% in total. A similar memory requirement improvement is found for the 1024*1024 case. The \(ST\) scheme requires the maximum of 200MB when all 14 nodes are utilized. Using 300 MB as the bottom line of allowed safety memory allocation, as suggested by Table 4.2, the system outrage should be avoided even if a full configuration of slave nodes is used.

#### Table 4.1: Memory requirement for \(ST\) scheme (MB) and the Ratio of Saving Compared to Original Scheme

<table>
<thead>
<tr>
<th></th>
<th>1 node</th>
<th>2 nodes</th>
<th>4 nodes</th>
<th>8 nodes</th>
<th>9 nodes</th>
<th>10 nodes</th>
<th>11 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tennis40</td>
<td>53.5</td>
<td>64.3</td>
<td>85</td>
<td>129.1</td>
<td>146.9</td>
<td>160.7</td>
<td>175.5</td>
</tr>
<tr>
<td></td>
<td>72.8%</td>
<td>61.7%</td>
<td>51.2%</td>
<td>44.7%</td>
<td>45.9%</td>
<td>45.8%</td>
<td>45.9%</td>
</tr>
<tr>
<td>Tennis60</td>
<td>47.5</td>
<td>54.2</td>
<td>77.6</td>
<td>110.4</td>
<td>120.1</td>
<td>130.8</td>
<td>141.5</td>
</tr>
<tr>
<td></td>
<td>79.8%</td>
<td>65.1%</td>
<td>59.4%</td>
<td>49%</td>
<td>48.2%</td>
<td>47.9%</td>
<td>47.6%</td>
</tr>
</tbody>
</table>

The effectiveness of our \(ST\) scheme is also confirmed by measuring the number of system page faults after necessary modifications are made. Figure 4.2 shows the...
average page fault caused by a slave node during the decompression of 60 frames per node. The same video files used in section 3 are tested. We observed that:

- The number of page faults is directly related to the picture size. The small size (352*480) has the lowest amount of page faults, while the high picture resolution (1404*960) corresponds to the highest page fault rate. This trend is also observed in Figure 3.4 for the original decoder.

- The number of page faults for each individual node is significantly reduced, comparing to the number in Figure 3.4. For the 1404*960 case, the number of page faults shrinks from 1500 to 1200, at one slave node configuration. For 1024x1024 video, the page faults are now 943, 25% less than before.

- For all of the video streams, the number of page faults almost remains unchanged when increasing the number of the slave nodes. This phenomena is also observed in Figure 3.2.b before the memory saturation point. The flat curves shows that the system memory usage is still under “control”. The page fault outrage for the two high resolution video streams are eliminated, showing that the ST scheme has successfully relieved the memory bottle neck.

![Figure 4.2: Memory Page Fault For The Revised Memory Management](image-url)
The obtained decoding frame rate gives a final judgment for the correctness of our analysis. Since our major target is the high quality video, we only show the accomplished frame rate for 1404*960 and 1024*1024 video streams. For each of the video sizes, we only show the accomplished frame rate for 1404*960 and 1024*1024 video streams. For each of the video sizes, we compare the performance for three video titles mentioned early. Figure 4.3.b show the scalable decoding frame rate for 1404*960 video of our revised ST scheme. We observed a close to linear increasing of the frame rate. For one slave node, we have 1.7 fps for tennis, 1.86 fps for flower, and 1.6 fps for the "mobil" video. For two slave nodes, the performance is nearly doubled for each case, with 3.5 fps for tennis, 3.7 fps for flower, and 3.2 fps for mobil. For the other node configuration, the achieved frame rate increase proportionally, and there are slight difference between the three video titles. The peak decode rates are obtained at 14-slave nodes, where 20 fps is observed.

The decoding performance for 1024*1024 video files shows a similar behavior. The close linear speed-up is also observed. At one slave node, the frame rate is roughly 1.8 fps. The highest frame rate is 23 fps for calendar when the system is fully loaded. The difference of frame rates for the three video title is slightly higher than that of 1404*960 size. The biggest performance difference occurred between calendar and tennis at 7 slave nodes, with a frame rate of 13 and 10 respectively. Nevertheless, the overall result still matches our expectation quite well, and there is no sign that the performance difference may increase.

For the two high resolution video formats, our revised ST scheme has successfully solved the memory shortage problem. The observed near real-time decoding rate shows that our scheme works well for high quality video up to MP@HL video. Our analysis of memory usage indicates that 14 slave nodes still leave enough memory space in the system. The theoretical maximum allowed slave nodes can
Figure 4.3: Decoding Frame Rate For the Revised Memory Management (a) 1024 * 1024 (b) 1404 * 960
be estimated as following: Still take 300 MB as the memory budget, we have $M_m = 42MB$ and $M_s = 11MB$ for the 1404*960 case. Thus the maximum number of slave nodes is $\frac{300-42}{11} \approx 32$. Given this amount of slave nodes, our scheme should produce up to a 45 fps decoding rate.

![CPU usage for tennis40](image)

**Figure 4.4: User Space Time VS Kernel Space Time for The First Buffer Optimization Scheme**

Figure 4.4 shows the overall CPU time distribution of slave nodes when decoding high-resolution video formats with the revised buffer scheme. The user space time component represents the computation time for the MPEG-2 decoding procedure, the kernel space time is for the system level overhead, including time spent in the network layer, system call, and other costs. It is observed that the user space time increases linearly when the number of slave nodes increase, accompanied by the counterparting decrease in the system idle time. Meanwhile the operating system level cost is maintained in a low level (between 5% to 10% of total CPU time). Particularly for the large scale experiments (more than 11 slave nodes)
deployed), the abnormality cross-over of the user space time and system overhead observed in the original decoding experiments no longer exists. This further proves the effectiveness of the ST scheme in solving the memory shortage.

### 4.3 Further Optimization in the Slave Nodes

In the above section, we use a 3-frame transmission buffer for each slave node, which has already obtained significant reduction in the memory requirements. We also showed that the memory shortage will not happen until there are more than 32 slave nodes participating in the parallel decompression, for the 1404*960 case. The maximum allowed number of slave nodes for other video sizes can be derived similarly based on the memory budget and the size of the video image. This result can be easily extended to the multi-layered MPEG-2 video, where an enhanced video stream layer may exist to feature SNR scalability, DATA partition scalable and other scalability features. For these scalable MPEG-2 streams, we usually have L-layer sub-stream, each requires the same amount of buffer space as that of the base layer in order to be successfully decoded. Assume a 3 layer MPEG-2 video is to be decoded; the decoding/transmission buffer in a slave node will be nearly tripled. It can be expected that the memory shortage will become a bottleneck again. To further reduce the buffer requirement in the slave node, we proposed a dynamic buffer requirement scheme.

It is observed that the decoding procedure in a slave node did not need three frames all the time. More specifically, the I-frame did not refer to any other frames, thus we can only use one frame as the decoding/transmission buffer. Similarly, P-frame only refers to one frame (I- or P-frame), thus the total need for decoding a P-frame is two. Only B-frame needs the whole three frame buffers. Thus the total amount of needed buffer can vary during the life time of the slave node. Since all the slave nodes are sharing the physical memory and are performing decompression
independently, we can effectively reduce the memory requirement by dynamically allocate buffers in the slave node.

Let the ratio of I, P, B frames in a GOP structure be \(a:b:c\), the effective buffer space for one layer is expressed by

\[
M = \frac{(1 \cdot a + 2 \cdot b + 3 \cdot c)}{(a + b + c)} \quad (4.1)
\]

In a typical GOP structure of "IBBPBBPBBPBBPBB", we have \(a:b:c=1:4:10\), this will result in an effective buffer number of \(39/15=2.6\), which is about 85% of the 3 frames buffer scheme. The effective buffer space is a function of the GOP structure. When the percentage of I frame increases, the effective buffer space will decrease. In the extreme case of all I-frame GOP, the effective buffer is 1 f/GOP. While a long GOP structure with many B frame will make the effective buffer space approaching the limit, which is 3 frame/GOP. Assume a two-layered scalable MPEG-2 stream with 1404*960 video size is to be decoded, the memory requirement of the slave node is \(M_s = 2 \cdot 11 \cdot 0.85 = 18.7 MB\). Still assuming a 300MB total memory budget and \(M_m = 42 MB\), the system can support up to \(\frac{300-42}{18.7} = 14\) slave nodes. Further assume the decoding time for such a two-layered high resolution stream is twice as high as the one-layer stream in a serial pure software decoder, the 14-slave nodes configuration can only produce up to \(14 \cdot 0.9 = 12.6\) frame/sec. In order to have a higher performance, we need further decrease the buffer space, such that more slave nodes can be supported. 

**Algorithm 1** below describes the dynamic buffer allocation scheme.

In fact, the concept of dynamic buffer allocation can be applied inside the decoding of each frame. Since the decompression of each frame is based on a serial decompression of macroblocks, the overall buffer space could be reduced by dynamically allocating buffer for macroblocks. For example, when decoding
Algorithm 1 Dynamic Buffer Allocation in Slave Node
/*Three Buffers outbuffer[1,2,3] are used repeatedly. */
/*forwardb, reverseb, and currentb point to the forwarding reference*/
/*frame, backward reference frame, and working frame respectively*/
/* The following steps decode and transmit one GOP frames */
RecieveGOP(&compressedBuffer)
allocate outbuffer[1,2,3]
for each frame $f(i)$ in compressedBuffer do
  if $f(i)$ is I-Frame then
    deallocate outbuffer[2,3]
    currentb=outbuffer[1]
    perform MPEG-2 I-frame decompression
    transmit the outbuffer[1] to master
  else if $f(i)$ is P-Frame then
    forwardb:=currentb, currentb:=outbuffer[2]
    perform P-Frame decompression
    transmit the outbuffer[2] to master
  else if $f(i)$ is B-Frame then
    reverseb=outbuffer[2]; currentb=outbuffer[3];
    perform B-Frame decompression
    transmit outbuffer[3] and release it.
end if
end for
de-allocate outbuffer[1,2,3]
the first macro-block, we only need allocate a 16*16 block space. The other macroblocks buffer will be assigned when it is needed for decoding. The total buffer will grow as more macroblocks are decoded, and will reach the maximum full buffer size after the decoding is finished. Then the slave will keep the full buffer size until the frame is able to be discarded. After the decoded frame is sent back to the master node, the decoding buffer can be released and a new buffer growing process will be started for the next frame. With this dynamic memory allocation, we can expect an additional buffer reduction of 0.5 frame for the current frame. Notice that this scheme can not reduce the amount of buffer for the reference frame, which should be in system through the whole process. The effective buffer requirement become

\[ M = \frac{(1 - 0.5) * a + (2 - 0.5) * b + (3 - 0.5) * c)}{(a + b + c)} \]

Using the same GOP structure as the above, the effective frame number of the buffer in slave node is 2.1, which is 60% of the 3-frame buffer scheme.

The tradeoff here is the additional CPU cost introduced for the dynamic memory management. For each macro-block, the additional cost includes at least two system calls (for memory allocation/deallocation) and some other miscellaneous operation. It has been showed that the cost associated with dynamic memory allocation is significant for the database server and Web-server, where thousands of processes may co-exist to processing user requests. In our case, the number of slave nodes/processes is usually below 20 and it is expected that memory management activity is far less frequent, thus the overhead introduced should be limited. This is confirmed by our experimental results by comparing the performance of the decoding with/without dynamic memory allocation, depicted by table 4.2. It can be seen that the increase of system time is very small. For 1024*1024 video format,
the system time with dynamic allocation enabled is 1.3 seconds, only 0.2 second more than the static memory allocation.

Table 4.2: Decoding Performance of Slave Node With Dynamic Memory Allocation
Measured Time in Second for The Total of Frames

<table>
<thead>
<tr>
<th>Video Resolution</th>
<th>500*360</th>
<th>704*480</th>
<th>850*750</th>
<th>1024*1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic User</td>
<td>4.37</td>
<td>7.02</td>
<td>10.61</td>
<td>12.08</td>
</tr>
<tr>
<td>Dynamic System</td>
<td>0.5</td>
<td>0.8</td>
<td>1.2</td>
<td>1.3</td>
</tr>
<tr>
<td>Original User</td>
<td>4.14</td>
<td>6.86</td>
<td>10.63</td>
<td>12.00</td>
</tr>
<tr>
<td>Original System</td>
<td>0.4</td>
<td>0.6</td>
<td>1.0</td>
<td>1.1</td>
</tr>
</tbody>
</table>
CHAPTER 5
CONCLUSION

Due to the limited memory to support a scalable performance for high-level high-profile MPEG-2 video resolutions, new buffering controls and mechanisms need to be created within our software-only parallel MPEG-2 decoder. We thus propose an ST buffering scheme with a dynamic allocation algorithm to significantly reduce the memory demands within this parallel decoding software. The results are very promising with excellent scalability performance achieved in both down-scaling and up-scaling capability. Therefore, it is now possible for our software-only parallel MPEG-2 decoder to automatically choose the best video resolutions (e.g., with proper number of slave nodes) according to the hardware and networking settings.
APPENDIX

/* Bounce - Creates a new thread each time the letter 'a' is typed. * Each thread bounces a happy face of a different color around the screen. * All threads are terminated when the letter 'Q' is entered. * * This program requires the multithread library. For example, compile * with the following command line: * CL /MT BOUNCE.C */

#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include "config.h"
#include "global.h"
#define MAX_THREADS 32

extern void frame_reorder(struct PictureBuffer *frame,int bitstream_framenum, int sequence_framenum);
extern motion_compensation (struct PictureBuffer *frame,short decsrc[6][64],int MBAMax,int MBA, int macroblock_type,
int motion_type, int PMV[2][2][2], int
motion_vertical_field_select[2][2],
int dmvector[2], int stwtype, int dct_type);

/* getrandom returns a random number between min and max, which must
be in
* integer range.
*/
#define getrandom(min,max)((rand())%(int)(((max) + 1)-(min)))+
(min))

//void main( void ); /* Thread 1: main */
//void KbdFunc( void ); /* Keyboard input */
//void BounceProc( char * MyID ); /* Threads 2 to n;display*/
//void ClearScreen( void ); /* Screen clear */
//void ShutDown( void ); /* Program shutdown */
//void WriteTitle( int ThreadNum ); /* Display title bar */

//HANDLE hConsoleOut; /* Handle to the console */
//HANDLE hRunMutex; /* "Keep Running" mutex */
//HANDLE hScreenMutex; /* "Screen update" mutex */
int ThreadNr; /* Number of threads started */
//CONSOLE_SCREEN_BUFFER_INFO csbiInfo; /* Console information */

/*
void mmain() // Thread One
{
    /* Get display screen information & clear the screen.

    */
int getframe(struct PictureBuffer * frame, int framenum)
{//resume the frame->data which is covered by mpi receive.
//this func call update_picture_buffers to attach a frame buffer.
//the relative point is fixed in the receiveDistributeData function.
{
    int MBAmax;
    int ret;
    int PsizeVerify;
    PsizeVerify=PARALLELSIZE;
    frame->data=global_microblocks[framenum%PARALLELSIZE];
// be overlapped by mpi transmit, the main process
// do not care it.
if (frame->picture_structure==FRAME_PICTURE && Second_Field)
{
    /* recover from illegal number of field pictures */
    // printf("odd number of field pictures\n");
    Second_Field = 0;
}
frame->Second_Field=Second_Field;

/* IMPLEMENTATION: update picture buffer pointers */
Update_Picture_Buffers(frame);

/* form spatial scalable picture */
/* ISO/IEC 13818-2 section 7.7: Spatial scalability */
if (frame->base.pict_scal && !Second_Field)
{
    printf("spatial_prediction, we don't support\n");
}

/* decode picture data ISO/IEC 13818-2 section 6.2.3.7 */

/* number of macroblocks per picture */
MBAmax = frame->mb_width*frame->mb_height;

if (frame->picture_structure!=FRAME_PICTURE)
MBAmax >>= 1;
/* field picture has half as many macroblocks as frame */

rMBA = 0;
frame->pnum = framenum;
frame->MBAmax = MBAmax;
for(;;)
{

    if((ret = slice(frame, framenum, MBAmax)) < 0) break;
    // if slice return -1, it mean we meet the start code
    for next picture.
}
// picture_data(frame, bitstream_framenum);

if (frame->picture_structure != FRAME_PICTURE)
    Second_Field = !Second_Field;

return 0;
}

void decodehighhalf(struct PictureBuffer * frame)
// decode the high half part of microblock for a picture
{
    int comp, i;
    int MBAmax;
MBAm = frame->MBAm;

for (i = (MBAm / 2); i < MBAm; i++)
{
    //for (comp = 0; comp < block_count; comp++)
    //    memcpy(ld->block[comp], frame->data[i].blocks[comp], 64);

    motion_compensation(frame, frame->data[i].blocks, MBAm, i,
                        frame->data[i].macroblock_type,
                        frame->data[i].motion_type, frame->data[i].PMV,
                        frame->data[i].motion_vertical_field_select,
                        frame->data[i].dmvector, frame->data[i].stwtype,
                        frame->data[i].dct_type);
}

void decodelowhalf(struct PictureBuffer * frame)
{
    //int comp;
    int i;
    int MBAm;
    MBAm = frame->MBAm;

    for (i = 0; i < MBAm / 2; i++)
    {
        //for (comp = 0; comp < block_count; comp++)
        //    memcpy(ld->block[comp], frame->data[i].blocks[comp], 64);
motion_compensation(frame, frame->data[i].blocks, MBAmax, i, 
    frame->data[i].macroblock_type, 
frame->data[i].motion_type, frame->data[i].PMV, 
    frame->data[i].motion_vertical_field_select, 
frame->data[i].dmvector, frame->data[i].stwtype, 
    frame->data[i].dct_type);
}
}

void bufferchange(struct PictureBuffer * srcframe, 
    struct PictureBuffer * desframe)
{
    desframe->ld->Incnt=srcframe->ld->Incnt;
    desframe->ld->Rdptr = (srcframe->ld->Rdptr-srcframe->ld->Rdbfr)
    +desframe->ld->Rdbfr ;
    desframe->ld->Rdmax = srcframe->ld->Rdmax;

desframe->ld->Bfr = srcframe->ld->Bfr;
memcpy(desframe->ld->Rdbfr, srcframe->ld->Rdbfr, 2048);
}

void dothework(char *MyId)
{

    printf("this is thread 2 running\n");
    // getch();
while(1)
{

if(decodestate==START)
{
    //WaitForSingleObject( hDecodeStart, INFINITE );
    //ResetEvent(hDecodeStart);
}
if(decodestate==MIX)
{
    //WaitForSingleObject( hDecodeMix, INFINITE );
    //ResetEvent(hDecodeMix);
    decodelowhalf(frameptr[(basenum)%3]);
    decodehighhalf(frameptr[(basenum)%3]);
    frame_reorder(frameptr[basenum%3],Bitstream_Framenum,
                   Sequence_Framenum);

    //getch();
    if (!frameptr[basenum%3]->Second_Field)
    {
        Bitstream_Framenum++;
        Sequence_Framenum++;
    }

    //SetEvent(hDecodeEnd);
    //WaitForSingleObject(hNEXTSTATE,INFINITE);
//ResetEvent(hNEXTSTATE);
} else if(decodestate==SINGLEDECODE)
{

//WaitForSingleObject( hDecodeSingle, INFINITE );
//ResetEvent(hDecodeSingle);
//decodehalf(frameptr[(basenum+1)%3]);
decodehighhalf(frameptr[(basenum+1)%3]);
//getch();
//SetEvent(hDecodeEnd);
//WaitForSingleObject(hNEXTSTATE,INFINITE);
//ResetEvent(hNEXTSTATE);
} else if(decodestate==END)break;
else if(decodestate==ERRHEAD)break;
}


%


REFERENCES


Yishu He was born in Tianjin, China. She received the Bachelor of Arts degree from Jilin University, China, in July 1998. She will receive her Master of Science degree in computer and information science and engineering from the University of Florida, Gainesville, in August 2002. Her research interests include MPEG-2 video compression.