A Real-Time Multimedia System for Video Applications

Gee-Gwo Mark Mei, Min-Hsiung Lin, Limin Hu, and Henry Chang

IBM T. J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

An embedded multimedia system is being constructed to provide high-quality video and audio processing capabilities for desktop multimedia workstations. This versatile JPEG-based system is capable of sustaining a throughput of 2 MBytes per second compressed data rate, which is a necessary condition for studio quality video and audio at the CCIR 601 resolution. The main challenge in design work is to methodologically evaluate critical data paths to ensure that the hardware system meets the periodic deadline of the isochronous multimedia data streams (e.g. 1/30 second per frame for motion video). We will describe in this paper the system design objectives, system architecture, and preliminary results of performance analysis.

1. Introduction

Many multimedia (MM) applications [1], such as video conferencing [2], interactive video [1, 3], graphic/voice animation [4], etc., require seamless integration of text, image, audio, video, and data communication in real time [5 - 9]. These applications demand very large bus bandwidth to transfer tremendous amount of data as well as sophisticated signal processing power to handle those data in real time.

Common solutions include high end computing facilities [10] or dedicated hardware. The cost of using high end computing facilities is sometimes prohibitive and they often require a special real time operation system (OS). Dedicated hardware, such as C-Cube 550 [11], provides real-time solutions for the JPEG baseline standard; however, using dedicated JPEG chips without carefully planning the integration of hardware and software may not be yield satisfactory results.

Many products on the market adopt the approach of using dedicated compression chips together with intelligent processor(s) to provide an inexpensive and yet versatile solution [12, 13]. Noticeable examples include DVI from Intel/IBM [14 - 16], Mware system from IBM, VCON multimedia development environment from AT&T [17], and SPOX from Spectrum Microsystems Inc. Despite the benefits gained from intelligent processor(s), more effort is required to study the whole system at once (i.e. all hardware and software components) so that cost effective solutions for providing studio quality multimedia systems can be achieved.

There are several multimedia hardware platforms as well as software authoring packages available in the market. For example, DVA-4000 from Video Logic, New Media Graphics' Super Motion Compression, Phoebe's VSA-1000, Dolch, and Visionary Motion JPEG are all based on C-Cube JPEG video compression processors. Rapid Technology's Visionary Motion JPEG card is a LS1 Logic based MJPEG compression board. Those JPEG image compression board can deliver image resolution from SIF to CCIR 601 at 30 fields per second to 30 frame per second. However, due to the limitations of system bus bandwidth, hard disk performance, and software system overhead, there is no product available that can deliver synchronized video and audio at the CCIR 601 resolution with 30 frame per second.

We take an integrated approach to build a real time multimedia system. First, design objectives are defined to draw the boundaries for our system. Second, multimedia data are treated as objects to describe system capabilities and application requirements. Then, a real-time multitasking Operating System methodology is used to effectively manage multimedia run-time support for both computation and I/O-intensive demands. Finally, mapping from multimedia object operations to the intelligent processors and dedicated compression chips are studied. In parallel, performance evaluation is conducted at all levels.

Section 2 of this paper presents our design objectives and Section 3 addresses our object oriented view. We address real time task scheduling issues and the mapping of object operations to system platforms in Section 4 and 5, respectively. Then in Section 6, an example of detailed implementation is shown. Section 7 looks into system integration and performance issues. Finally, Section 8 summarizes this paper and discusses future work.

2. System Design Objective

Our system design objective is to construct a studio quality, versatile, and flexible embedded multimedia system for playing back, editing, or conferencing with remote or local entities through multiple high-resolution multimedia windows with live motion video. The external input sources considered are video and audio data streams. Typical data rate of digital audio ranges from 8 KBytes/second (telephone quality voice) to 176.4 KBytes/second (CD music quality, i.e., 44.1 KHz sampling rate, two channels, and 16 bits per sample). On the other hand, video rates are much higher, and they often exceed current system limitations. Consider the data rates for the following motion video qualities at 30 frames/second:

<table>
<thead>
<tr>
<th>Display</th>
<th>Resolution</th>
<th>1 Byte / pixel</th>
<th>2 Bytes / pixel</th>
<th>3 Bytes / pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td>XGA</td>
<td>1024x768</td>
<td>22 MB/s</td>
<td>44 MB/s</td>
<td>66 MB/s</td>
</tr>
<tr>
<td>CCIR 601</td>
<td>720x480</td>
<td>10 MB/s</td>
<td>20 MB/s</td>
<td>30 MB/s</td>
</tr>
<tr>
<td>VGA</td>
<td>640x480</td>
<td>9.2 MB/s</td>
<td>18.4 MB/s</td>
<td>27.6 MB/s</td>
</tr>
<tr>
<td>VCR</td>
<td>352x240</td>
<td>2.6 MB/s</td>
<td>5.2 MB/s</td>
<td>7.8 MB/s</td>
</tr>
<tr>
<td>PLV (DVI)</td>
<td>512x480</td>
<td>7 MB/s</td>
<td>14 MB/s</td>
<td>22 MB/s</td>
</tr>
<tr>
<td>RTV (DVI)</td>
<td>256x240</td>
<td>1.8 MB/s</td>
<td>3.5 MB/s</td>
<td>5.5 MB/s</td>
</tr>
</tbody>
</table>

Table 1: Display Resolution versus Data Rate

The number of bytes for a single pixel depends on the video coding format. For example, the RGB24 format needs 3 Bytes/pixel, and the YUV16 format needs only 2 Bytes/pixel. A 256 mapped color format requires only 1 Byte/pixel. Clearly, the bandwidth shown in Table 1 exceed or approach to the bandwidth limit of almost all personal computer systems. Therefore, applying video compression techniques to handle the high video bandwidth becomes critical for most video applications.

The compressed JPEG bandwidth versus various row display resolution is exemplified in Table 2. In other words, if we consider using JPEG compression algorithms, then either full NTSC or VGA resolution requires close to 2 MB/s compressed bandwidth. Based on this design criterion, we are able to determine the bandwidth requirement of this system. (Even for CD quality, the bandwidth consumed by audio after compression is considerably less than that consumed by video). In the following sections, we will discuss our application domains based on an
Let us take a products-oriented view enables us to present the system structure in a hierarchical way. For example, a user can issue a command to bring the window 1 to the front as shown in Figure 1(b). The operations to complete the user instruction are described as follows. Let us assume that at time $t_0$, window 3 was displayed on the top of window 1. After a user command was issued at time $t_1$, to bring the window 1 to the front, the window object receives a message to repaint. Then the clip object is invoked to play back the video and audio clip based on the new overlay structure (window 1 in front). Then the video and audio frame object are invoked to be read, and decompressed. Finally, the video streams are composed with audio streams and displayed on output devices. That means after time $t_2$, the new window arrangement is displayed. During the period from time $t_1$ to $t_2$, the real time OS of the embedded system will off load the timing critical tasks from the PC OS and dispatch them. Task scheduling by the real time OS are to be discussed in the next Section.

4. Real Time Scheduling

We refer again to the example shown in Figure 1(b) and (c) in order to illustrate the real time task scheduling for the object operations. We divide the tasks into three categories as shown in Figure 2. There are the tasks set $T_{all}$, $T_{al}$, and $T_{a}$. The interrupt tasks are only allowed at a certain period of time during each frame time. The tasks in the $T_{al}$ set are:

1. Task $T_{a1}$: read video frame $v_1$.
2. Task $T_{a2}$: read video frame $v_2$.
3. Task $T_{a3}$: read video frame $v_3$.
4. Task $T_{a4}$: read audio frame $a_3$.

There are no task dependency among these tasks. Therefore the real time OS can schedule these tasks in any order. However these tasks should end after a time period $K$ where $K$ is the execution time defined in (1).

$$\sum_{i=1}^{2} Time(T_{al}) + Time(T_{a}) = K$$

![Figure 2](image)

Figure 2: Task Dependency Graph for the Example

The tasks in the $T_{al}$ set are:
1. Task \( T_{v4} \): decompress video frame \( v_1 \).
2. Task \( T_{v5} \): decompress video frame \( v_2 \).
3. Task \( T_{v6} \): decompress video frame \( v_3 \), and
4. Task \( T_{a1} \): read audio frame \( a_1 \).

The total execution time for this task set is defined by

\[
\sum_{i=4}^{6} \text{Time}(T_{ai}) = L_4 \tag{2}
\]

The tasks in the \( T_{a2} \) set are:
1. Task \( T_{a3} \): decompress the audio frame \( a_1 \).
2. Task \( T_{v7} \): composite the video frame \( v_1 \), \( v_2 \), and \( v_3 \) with audio frame \( a_1 \).

The total execution time for this task set is defined by

\[
\text{Time}(T_{v7}) + \text{Time}(T_{a3}) = L_2 \tag{3}
\]

The tasks in the \( T_1 \) are the interrupt routines that communicate with the host system. The execution time for the interrupt routines is \( L_1 \). According to Figure 2, the execution time \( K + L_1 + L_2 + I \) must be less than 1/30 second (or one frame display time). However, this is only a transitional constraint which affects only the transitional buffer storage size. The steady state constraints in (4) should be less than one frame period, i.e., processing time per frame (e.g., 1/30 second).

\[
K + L_1 + L_2 \leq \text{Frame Period} \tag{4}
\]

5. System Architecture

The object operations need to be mapped to a particular hardware platform. For example, the task \( T_{v4} \) reads data from a large video store (hard disk) to a hardware CODEC subsystem. Direct access between hard disk and the hardware CODEC subsystem implies special software and hardware arrangement, which is not available on all platforms and not considered by this paper. We read data from hard disks to system memory and then to the CODEC subsystem. The rest of the task allocation is shown in Table 3.

<table>
<thead>
<tr>
<th>Task Id</th>
<th>Actions</th>
<th>Allocation 1</th>
<th>Allocation 2</th>
<th>Allocation 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_{v1} )</td>
<td>Read video frame ( v_1 ) of Window 1</td>
<td>hard disk</td>
<td>memory</td>
<td>CODEC</td>
</tr>
<tr>
<td>( T_{v2} )</td>
<td>Read video frame ( v_2 ) of Window 2</td>
<td>hard disk</td>
<td>memory</td>
<td>CODEC</td>
</tr>
<tr>
<td>( T_{v3} )</td>
<td>Read video frame ( v_3 ) of Window 3</td>
<td>hard disk</td>
<td>memory</td>
<td>CODEC</td>
</tr>
<tr>
<td>( T_{a1} )</td>
<td>Read audio frame ( a_1 ) of Window 3</td>
<td>hard disk</td>
<td>memory</td>
<td>CODEC</td>
</tr>
<tr>
<td>( T_{i} )</td>
<td>Receive interrupt from host</td>
<td>PC CPU</td>
<td>CODEC</td>
<td></td>
</tr>
<tr>
<td>( T_{a2} )</td>
<td>Read audio frame ( a_2 ) of Window 1</td>
<td>hard disk</td>
<td>memory</td>
<td>CODEC</td>
</tr>
<tr>
<td>( T_{v4} )</td>
<td>Decompress ( v_1 ) of Window 1 (( w_1 ))</td>
<td>hard disk</td>
<td>memory</td>
<td>CODEC</td>
</tr>
<tr>
<td>( T_{v5} )</td>
<td>Decompress ( v_2 ) of Window 2</td>
<td>hard disk</td>
<td>memory</td>
<td>CODEC</td>
</tr>
<tr>
<td>( T_{v6} )</td>
<td>Decompress ( v_3 ) of Window 3</td>
<td>hard disk</td>
<td>memory</td>
<td>CODEC</td>
</tr>
<tr>
<td>( T_{a3} )</td>
<td>Decompress ( a_3 ) of ( w_1 )</td>
<td>hard disk</td>
<td>memory</td>
<td>CODEC</td>
</tr>
</tbody>
</table>

5. System Architecture

The object operations need to be mapped to a particular hardware platform. For example, the task \( T_{v4} \) reads data from a large video store (hard disk) to a hardware CODEC subsystem. Direct access between hard disk and the hardware CODEC subsystem implies special software and hardware arrangement, which is not available on all platforms and not considered by this paper. We read data from hard disks to system memory and then to the CODEC subsystem. The rest of the task allocation is shown in Table 3.

<table>
<thead>
<tr>
<th>Task Id</th>
<th>Actions</th>
<th>Allocation 1</th>
<th>Allocation 2</th>
<th>Allocation 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_{v1} )</td>
<td>Composite ( v_1 ), ( v_2 ), ( v_3 ) so that ( v_1 ) will be on the foreground</td>
<td>CODEC</td>
<td>video and graphic controller</td>
<td>display</td>
</tr>
</tbody>
</table>

Table 3. Task Allocation for the Example in Figure 1

Figure 3 is the system architecture on which we map all the intended object operation to hardware. The subsystems in the system architecture are described as follows:

1. Capture module which captures and digitizes video/audio streams;
2. Compression/Decompression (CODEC) module which compresses digital video/audio streams;
3. Display module that composes and displays the video, graphic, and text objects;
4. Hard Disk Storage subsystem stores the compressed video/audio streams to be used for playing back by the local or remote hosts;
5. Communication subsystem transfers data from/to local/to/from remote stations;
6. Graphic subsystem that controls and displays the video and graphic entity;
7. Host system that includes CPU and system memory and coordinates all system resources.

System Bus

Figure 3. System Configuration

Two types of constraints, processing time (such as equation (4) and bandwidth (due to studio quality), are inherited from the previous analysis. In order to enforce the processing time constraints by the host OS, the host OS must either support deadline processing [5], or accommodate additional kernel-level processing modules. For example, resident programs on DOS and loadable kernel extension on AIX V.3 [21] allow users to add modules into the kernel to do time-critical processing. The other approach is to put real time constraints on the CODEC subsystem. The task allocation is modified when finer hardware allocation is defined. Two revised allocations are shown in Table 4 for illustration purpose.

<table>
<thead>
<tr>
<th>Task Id</th>
<th>Actions</th>
<th>Allocation 1</th>
<th>Allocation 2</th>
<th>Allocation 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_{v1} )</td>
<td>Read video frame ( v_1 ) of Window 1</td>
<td>system interface</td>
<td>on board processor</td>
<td>SRAM</td>
</tr>
<tr>
<td>( T_{v4} )</td>
<td>Decompress ( v_1 ) of Window 1 (( w_1 ))</td>
<td>Video CODEC chip</td>
<td>frame store</td>
<td>FPGA state machine</td>
</tr>
</tbody>
</table>

Table 4. Revised Task Allocation in CODEC Subsystem

After we revised allocations such as the those in Table 3 and other object operations, the CODEC module contains the following hardware components:
(1) System bus interface logic which interfaces with the host system to exchange control, status information and supports high speed data movement between the host and the CODEC module.

(2) Compression/Decompression engine which compresses/decompresses video/audio streams according to the standards such as JPEG, MPEG for video and MP3 for audio streams [22], G.711, and G.722 for audio streams [23].

(3) Microprocessor which is equipped with real time multitasking capability to perform task scheduling such as video and audio synchronization [5, 6, 24].

(4) Capture/Display interface logic which is responsible for taking in video/audio streams for compression or sending out decompressed video/audio streams for display.

(5) Compressed and decompressed data buffer for regulating the data flow among the components for the CODEC module.

The processing time constraints are also imposed on this level. However, the constraints equations may need to be modified based on the new allocation. For example, in equation (2), we assume that \( T_{V4}, T_{V5}, T_{V6}, \) and \( T_{A1} \) run in parallel. Since dedicated hardware is used to decompress video, the process \( T_{V4}, T_{V5}, T_{V6} \) can be performed in parallel with task \( T_{A1} \). In the next Section, we will describe in detail the analysis of the steady state processing time and bandwidth constraints based on the reallocated allocation.

6. Object Operation Mapping

We use compressed video streams to illustrate the mapping of this object model to hardware implementation. Decompression process is the same except that the direction of the data movement is reversed. Without loss of generality, we use one full screen video stream for analysis instead of three streams. We do not allow multiple video streams to exceed the full screen.

In the capture/compress mode, a unit of work is called a clip, which contains a sequence of frames. The embedded system receives, compresses and stores a clip of frames in pipeline. The data movement of the pipeline must meet the 1/30 second per frame deadline for the integrity of the clip. The compressed clip is stored in consecutive disk blocks. In order to facilitate random access for individual frames, we employ a separate index file to store the location and the byte count of frames. The file structure in Table 5 shows the data organization on hard disks.

<table>
<thead>
<tr>
<th>Index File</th>
<th>Data File</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_start_address</td>
<td>v_byte_count</td>
</tr>
</tbody>
</table>

Table 5: Compressed Data File Types

The operation flow of compressing a clip is described as follows:

**Host System**

**Start:** The user application initiates a clip compression with the specification of start_of_the_clip, and end_of_the_clip (in time or frame counts);

**Preparation:**
1. Pre-allocate memory buffers for staging video data and audio data before they are stored onto disks.
2. Pre-allocate disk storage file for both the frame data and index data.

**Start the adapter microprocessor:**
1. Instruct adapter microprocessor to start the compression of a clip.
2. Supply to the adapter a list of free buffers and the expected frame count.
3. Enable the frame-ready interrupt from the adapter microprocessor.

**Synchronous data movement (must respond within 1/30 second):**
When receiving a frame-ready interrupt (the CODEC driver):

**Do Begin:**
1. Read the byte-count of the current frame and prepare the indexing information.
2. If current frame_count ≤ clip_frame_count re-enable the interrupt (within 1/30 sec)*
3. Initiate transfer of complete data blocks to disk
4. Add buffers freed by completion of disk transfer to the adapter's buffer reservoir**

**End;**
* The interrupt priority must be set such that the deadline can be met.
** If the disk storage I/O traffic is heavy, there is a danger of buffer exhaustion here. Allocate more buffers to smooth out intermittent load fluctuation.

The basic idea and underlying assumptions are stated in the following to demonstrate that synchronous data movement can be done in 1/30 second. First a simple G/G/1 queue concept is introduced as the model for us to derive the proof. The host system takes data from CODEC board and stores them in the system memory. Then the CPU of the host tries to pack the video and audio data and send them to a hard disk. To prove that the host system can respond within 1/30 second, we simply have to show that the queue will be in steady states once it enters some steady state.

The following are the basic assumptions:
1. The data producing rate by the CODEC subsystem is defined to be 720 x 486 x 2 x 30 / 10 = 2.1 M bytes/sec (assume 10 to 1 compression rate for video) plus 44.1K x 2 x 2 / 2 = 0.09 M bytes/sec (assume 2 to 1 compression rate for audio) which is total 2.19 MBytes/sec and 7.3 KBytes/frame.
2. A hard disk is dedicated to store video and audio data, so the disk head movement (seek time) of the hard disk is minimized.
3. The bandwidth of the hard disk subsystem is 2.2 MByte/sec.
4. The CPU in the host runs on a 25 MHz clock. It takes 4 cycles to complete one instruction. The number of instructions to take the data from CODEC and send them to hard disk is 5K, which translates into 0.8 ms in elapsed time.
5. The system bus transfers 2 bytes of data in 200 ns. However, the host system reserved 40% of the rights to use the system bus. The rest of the bus bandwidth are reserved for data transfer from CODEC to system memory and from system memory to hard disk. This means that it takes 24.3 ms to transfer the data from CODEC to hard disk during a frame period (1/30 second).
6. The host CPU copies all video and audio data to a contiguous location for data transfer from system memory to hard disks. The system memory can transfer data with 4 bytes per 100 ns. Therefore, it requires 3.6 ms to move a frame out of 50% utilization bandwidth.

On the basis of those assumptions, it requires on the average a total processing time of 28.7 ms, which is 3.6 ms + 0.8 ms = 4.4 ms plus data transfer time of 24.3 ms. Since each frame comes at 30 frames per second, that is, 33 ms per frame, a system with 28.7 ms service time is stable in steady states.
System Memory

Figure 4. G/G/1 Queue

**Adapter microprocessor**

**Start:** Be waken up by the host's compression request

**Preparation:**
1. get the start/end clip information from the host
2. get the list of buffers (pages) from the host
3. instruct CODEC engine to be ready for compression
4. instruct video and audio capture module to start capture
5. enable compressed-buffer-ready interrupt by the CODEC engine

**Synchronous data movement:**

Do While compressed-buffer-ready interrupt occurs
1. get video and audio data starting address and byte_count from CODEC engine
2. reset the flag to enable the CODEC engine to start on next frame
3. while Not (end of frame) OR (transferred count < byte count)
   DO Begin (for one transfer unit)
   a. check for available free buffer space in the system memory
   b. initiate data transfer from adapter internal buffer to the system memory
   End
   If end of frame Then
   a. interrupt host system with total byte-count, buffer address, and status
   b. receive update of free buffers
   EndIf
4. return to wait for the next compressed-buffer-ready interrupt
End When;

The compressed clip may be transferred directly to the SCSI device if there exists a direct path between adapter board and SCSI storage. This alternative is good on low-end PC where the host system but is not capable of sustaining data movement of adapter-memory-disk without locking up the whole system.

**Dedicated hardware logic (CODEC engine)**

**Start:** The adapter microprocessor resets the CODEC engine for compression start; the adapter microprocessor starts the capturing logic;

**synchronous data movement:**

while current frame ≤ end_of_the_clip

**While not end of current frame**

//capture module //

1. video data line: capture, compress and store in video buffer
2. audio data line: capture, compress and store in audio buffer

// buffer management logic //

3. If end of frame Then interrupt micro-processor; // buffer control //

(give micro-processor the pointers of v_data, a_data, v_byte_count, a_byte_count)

End While;

**Note step 1, 2, 3 are executed concurrently by the individual hardware modules.**

Performance analysis for dedicated hardware and subsystem processor indicates that they show satisfactory performance. Therefore, data transfer between SCSI storage subsystem and system memory becomes critical and we will discuss these issues in the next section.

**7. System Integration and Performance Issues**

The main system bottleneck we encountered for the high quality video editing scenario is the DASD I/O throughput. A disk access time consists of seek time, rotational latency, data transfer time, and controller overhead. Because of the sequentially property of multimedia data, disk head seek time can often be minimized. The effect of rotational latency can also be mitigated by large block transfers, as shown in Figure 5. Therefore, the data transfer rate is the source of the I/O throughput limitation.

Figure 5 demonstrates the DASD bottleneck and possible solutions. Three different SCSI disks with different capacities running under OSR 2.0 are tested. None of the disks we tested provides adequate I/O throughput. The maximum throughput for each drive is much less than the specified data transfer rate, which is calculated only from physical disk data layout and spindle rotational speed.

When we utilize parallel disk accesses to multiple disks under a single SCSI host adapter, disk I/O throughput can be increased to more than 2 MB/s using various techniques with large block sizes. Four methods are used to mix two disk 3 in Figure 5:

1. Round Robin: Disk accesses are issued from the disk device driver to the SCSI host adapter in a round robin fashion.
2. Semaphore-Synchronized DosReadAsync: A single-threaded process issues asynchronous I/O requests to different disks. However, the process uses semaphore to coordinate different drives such that a new round of requests will only be sent after all drives have finished the previous round of requests.
3. Asynchronous DosReadAsync: A single-threaded process issues a new request for any drive that has finished the pending request.
4. Multi-Threaded Asynchronous: Several threads run at the same time, and all the I/O requests to one disk are issued from a single thread.

Significant improvement of overall I/O throughput is observed from any of the above four methods. In the case of heterogeneous disk mixing, more modification is needed to achieve high I/O throughput [25].

Another way to break this bottleneck is to use a single disk with better performance. We observe an I/O throughput of more than 2 MB/s from one Micropolis disk on an IBM PS/2 Model 95. The performance measurement is shown in Figure 6. Only a small operating region provides sufficient I/O throughput.

Based on the information that the CODEC subsystem supports a minimum of 2 MB/s decompression rate, and the average JPEG compressed object requires 1.5 bit/pixel, the system can display live video in 720x480 resolution, with enough residual bandwidth for high quality audio. However, one high quality DASD drive on an high end personal system with well controlled environment, or two DASD devices in parallel is required to support this data rate. Also, the process to retrieve objects from the main memory is random, and some frames have to be dropped to support this kind of resolution.

\[1\]Not all the disk device drivers support parallel disk accesses. For example, DOS and OS/2 1.3 disk device drivers do not issue multiple I/O requests to the storage subsystem at once. On the other hand, OS/2 2.0 supports parallel disk accesses.
For example, in order to support 2MB/s data rate, it would require more than ten CD-ROM drives in parallel. For a JPEG type compressed video object with 1.5 bit/pixel, the video quality displayed by the subsystem is about or less than DVI RTV quality. MPEG2 type algorithms that take into consideration of motion compensation is needed to further increase the compression ratio for CCIH 801 quality video.

8. Conclusion

This paper describes our methodology for constructing a high quality, flexible, integrated multimedia system. Advances in semiconductor technology and availability of high power CODEC and microprocessor chips enable us to build an intelligent subsystem for sophisticated multimedia applications. Playing back, editing, or conferencing with remote or local entities through multiple high resolution multimedia windows in live video mode is expected to appear in the near future, provided that an integrated approach is used to attack real time multitasking scheduling, resource allocation, and system integration issues at once. The object oriented approach offers benefits of software reuse to reduce the development effort. However, it is laborious to keep track of the timing constraints and resource allocation. Further tool research and development are still much needed for the integration of the multimedia hardware and application.

9. Reference