Multimedia Systems (2003)
Digital Object Identifier (DOI) 10.1007/s00530-003-0121-4

Multimedia Systems


# Internet-based interactive HDTV

**Bin Yu, Klara Nahrstedt**

Department of Computer Science, University of Illinois at Urbana-Champaign, 1401 West Green Street, Urbana, IL 61801, USA
(e-mail: binyu,klara@cs.uiuc.edu)

Received: ♣ / Accepted: ♣

**Abstract.** Traditional interactive TV systems depend on expensive hardware, proprietary formats, and a closed-loop end-to-end approach, which greatly limits scalability and extensibility of TV services. In this paper, we present the *HDControl* interactive Internet TV architecture that achieves an open service model and combines high-quality video with flexible user control using two key software real-time algorithms: visual information embedding (VIE) algorithm and resynchronization algorithm. Experimental results in our HDTV testbed have confirmed the feasibility and efficiency of our proposed algorithms.

## 1. Introduction

Recent years have seen many advances in several fields that have generated new results in multimedia information distribution technology. MPEG2 standard [3] has become the most widely accepted standard for video transmission and storage in various video applications, such as the standard-definition TV and high-definition digital TV (HDTV) [5] broadcast, Internet video-on-demand, interactive TV, and video conferencing. On the other hand, broadband networking technologies such as cable modem [8] and xDSL [6] are bringing to our homes and office buildings Internet connections with up to 20 Mbps or higher bandwidth at acceptable prices. More advanced technologies of even higher resource availability, such as Fast Ethernet [13] and Gigabit Ethernet [2], are providing capacity to afford multiple high-quality video streams simultaneously. At the same time, high-end personal computers are becoming more powerful with processors of speeds over 3 GHz, large memory, and capacious storage devices. Finally, tremendous amounts of information resources have appeared on Internet servers (e.g., yahoo.com), which are becoming the most popular model of information sharing.

With all these exciting technologies, we can expect much better video distribution services offering higher visual quality, richer information content, greater interactivity, more flexibility, and finer customization. Aiming at the same goal, two originally separate approaches, *PC plus Internet* and *TV cable network plus set-top box* , are rapidly converging to provide a common set of services including pay-per-view, multichannel view via picture-in-picture (PiP), Web browsing, stock/weather/sports/news updating, e-mail accessing, and others. The first approach, PC plus Internet, provides easy interactivity, flexibility, and customization of various multimedia services (Fig. 1a). But this approach does not provide high-quality HDTV display and processing due to limited screen size and lack of efficient software solutions that are capable of composing high-bitrate MPEG2 video streams online.[1] On the other hand, the second approach, TV cable network plus set-top box, provides high-quality video delivery through special hardware at the TV devices or set-top box (Fig. 1b). However, this does not support easy interactivity, flexibility, and customization of the displayed content since all the video processing is done in a closed world – from the TV station to the set-top box – using predefined operation modules and proprietary formats.

Therefore, we need a third approach that combines the good features of the previous solutions in a new architecture. In this paper, we present one solution that allows for high-quality content delivery, easy interactivity, flexibility, and service customization (Fig. 1c). Our solution uses TV devices for high-quality display, but all the content will come from the Internet. To allow this to happen, we need to put PCs on the path between the TV and Internet sources due to the format and protocol mismatch. The PCs function as smart and open set-top boxes that allow for interactivity between the user and Internet, customization and flexibility of services, and content delivery of a video stream to the TV devices in an appropriate format. To enable such an architecture, it is crucial to have an open, flexible, and scalable software solution to compose multiple MPEG2 video streams online. The key vision is that all the services mentioned above, such as Web/e-mail browsing and news updates, can be implemented through a *visual information embedding (VIE)* process. VIE refers to an operation that embeds any visual information (video, image, or text) into the original video stream. Example applications include *content embedding*, such as picture-in-picture (PiP), logo/ticker insertion and captioning, and *control interface presentation*, such as displaying menus and buttons on the TV screen to invite

[1] The TV networks have agreed to deliver all digital video content in MPEG2 format.

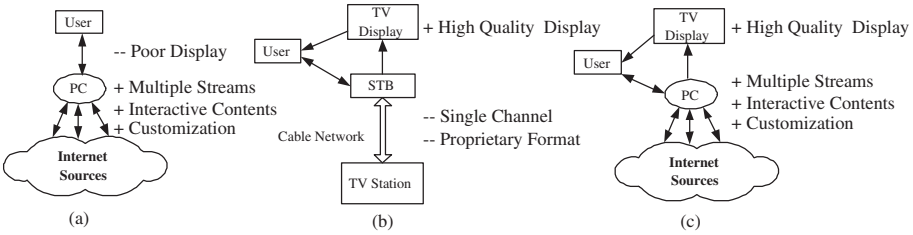


**Fig. 1.** Three models for interactive TV

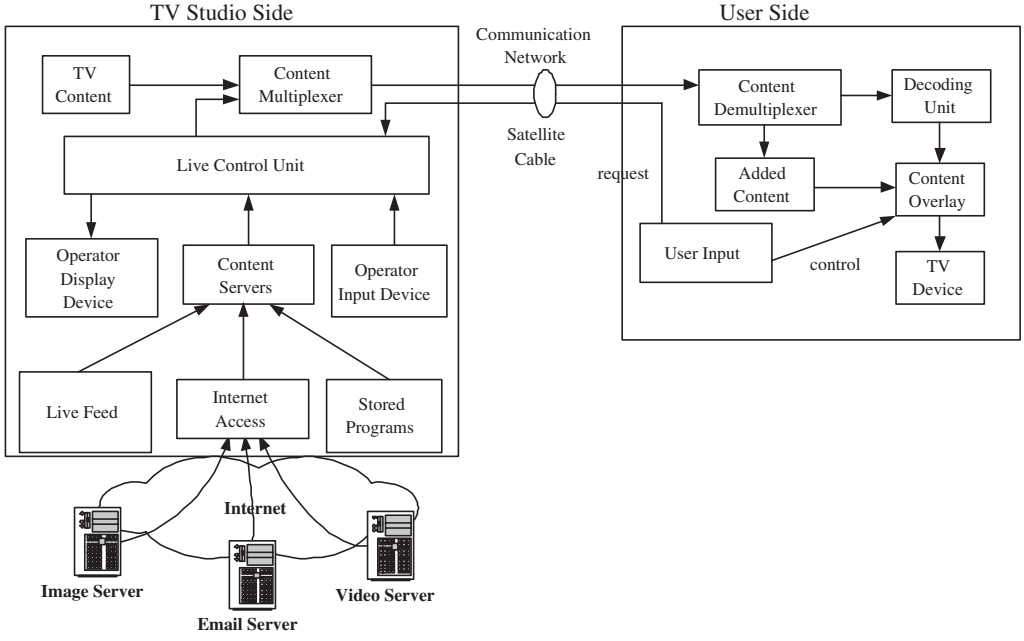


**Fig. 2.** Traditional interactive TV architecture

the user's interaction. VIE allows us to *edit* video streams by embedding data information retrieved from the Web and control information from user commands onto the background MPEG2 video stream.[2]

Our solution has many challenges and benefits (Fig. 1c). One challenge is that PCs are far from fast enough to naively decode and then encode each frame to embed new content. Another challenge results from the loss of synchronization due to video editing operations. The benefits of our solution are also manifold: it gives users more flexibility and control over how the service is customized to their needs, it provides an open and standard platform for third-party service providers to offer content and functions, and it gives software programmers more control over the presentation of multimedia information.

In the next section, we will present our HDControl interactive Internet TV architecture by comparing it with traditional interactive TV approaches. In Sects. 3 and 4, the two key algorithms that enable real-time video manipulation are discussed. Section 5 presents experimental results, and Sect. 6 concludes the paper.

[2] In contrast to the windowing mechanisms used for PCs, TV decoding devices only accept a single MPEG2 standard formatted stream, which is why we need to make sure the output from the PC is formatted in this way.

## 2. Interactive TV

To better illustrate the advantages of our architecture, we first describe traditional interactive TV and then present our new solution.

### *2.1. Traditional interactive TV architecture*

Figure 2 is a simplified diagram of a traditional interactive TV solution that provides live data insertion and distribution (see [11] for a more detailed description). Basically the primary TV content in MPEG2 format is multiplexed with the added content and sent from the TV studio through the communications network (such as cable lines or satellite signals) to user devices, where it is demultiplexed, decoded, and displayed on the TV device. User input controls how the added content should be overlaid on top of the decoded TV content, and it may also trigger some requests that are routed back to the studio's live control unit. In the studio, user requests are displayed to the operator with the operator display devices, and the operator will specify what kind of new content is needed (e.g., stored video programs, live content feed, or data obtained from the Internet) and how it should be multiplexed with the current TV content.

Though this model has proven to be practical and efficient, its closed-loop end-to-end approach inevitably leads to several deficiencies:

1. **Scalability.** When the user subscribes to more services and requests more content to be inserted, all the data have to be sent back to the studio first and all the processing has to be done inside the studio. This is not scalable because it increases processing and scheduling complexity at the live control unit and creates traffic concentration at the Internet access.
2. **Compatibility.** Because the TV program is sent through a closed channel in proprietary format, different vendors' TV programs are not compatible. Third-party service providers cannot fit into this picture other than by providing data to the content server in the TV studio. Also, if users want to switch to a new TV service provider, they have to buy a new set of client devices, which is inconvenient and expensive.
3. **Customizability.** Since the content multiplexing and embedding are performed in a predefined, nonstandard end-to-end manner, it is hard for users to arbitrarily customize the content insertion and information overlay. For example, the viewer cannot access e-mail *while* watching TV, and the position and size of the embedded window is also limited to predefined settings.

### *2.2. HDControl interactive Internet TV architecture*

Figure 3 illustrates our new architecture for interactive Internet TV. Because it supports high-definition television (HDTV) streams yet allows users greater control over how the TV stream is presented, we give it the name *HDControl*. The Active Service model [4] is adopted in which video editing *servents* are deployed across the Internet to provide data forwarding and video editing services such as VIE and bitrate control. The primary data flow is the following: the TV content encoded in standard MPEG2 format is forwarded through the high-bandwidth Internet connection to the *user computer* and then decoded and displayed on the *TV set*. On top of this, the commands from user interaction and information from the *home network* are handled by a VIE processing module at the user's computer. External content can be provided by any data server on the Internet and added to the primary TV data stream at the VIE servents along the path.

Because the Active Service model has been widely used, we will only briefly describe how our VIE service fits into it. Initially, no service is subscribed to, and the TV content will go through the IP network directly to the user's computer. When the user specifies a new piece of information to be added, the *user interaction module* will contact a *bootstrap server* to identify the nearest VIE service cluster along the streaming path and initiate a new VIE servent in that cluster. This new servent will then set up a connection with the data server that provides the content requested by the user. At this point, the TV stream will be rerouted through this new servent, and the data server will begin to send the content to the servent, who embeds it into the TV stream. Whenever another new content is requested, either an existing VIE servent is reused to set up the connection with that data server or a new servent is created along the path.

In summary, the special features that distinguish our architecture from the traditional one are:

**Standard video format:** MPEG2 transport format is used across all processing and communication until finally the program is decoded and displayed on the TV device, and both HDTV and standard-definition TV (SDTV) are supported. Our architecture allows the TV station's data server, the intermediate service / content providers, and the user devices to be implemented independently and cooperate without any compatibility problems. Also, the bitrate of the stream becomes much more controllable because the inserted information is embedded directly instead of being carried additionally along the stream.

**Distributed data server:** As shown in Fig. 3, the data servers are now distributed across the whole Internet instead of being confined within the TV studio. Each server can specialize in a particular kind of information, such as video, audio, image, e-mail, Web pages, stock values, weather reports, etc. Users can request different types of information, and the control units of each data server will respond to these requests and send the specified content to the nearest VIE gateway on the path of the primary stream. In addition, at the user's home the information sources from the electronic devices in the *home network* can also be formatted into visual content and embedded onto the TV stream when necessary.

**Distributed data processing:** The embedding of different information from different data servers happens in a distributed fashion along the path of the stream to avoid central failure and reduces complexity at each VIE gateway. When several data servers are close to each other, their contents can be first merged on their way to the stream path by intermediate VIE gateways as well, as illustrated by the Weather Server and the Database Server in Fig. 3.

**Flexibility and customizability:** Because all kinds of data/control information are directly embedded into the TV content using software modules, our TV service provides much greater flexibility for user control and customization. For example, users can use any input devices that communicate with a user PC, and the type of the information embedded can be flexibly specified and changed at no cost. For instance, when a user wants to view her e-mail in a separate TV window while watching a football game, she can specify the size and position of the e-mail window or even the font of the characters.

The key result that enables our architecture is the ability to do (1) real-time software VIE on MPEG2 streams and (2) resynchronization after video editing operations. We will discuss our algorithms in detail in the next two sections.

## 3. Visual information embedding

A naive solution is that each VIE servent first decodes the stream into raw pixels, then overwrites the embedded content on top of the background pixels, and finally reencodes the raw data back to MPEG2 standard format. Obviously this would be very expensive and slow with the current capabilities of hardware even for standard-resolution digital TV (as opposed to HDTV). Therefore, we have developed a suite of two key algorithms that embed visual information directly into the MPEG2 compressed stream in real time (see below) and adapted the

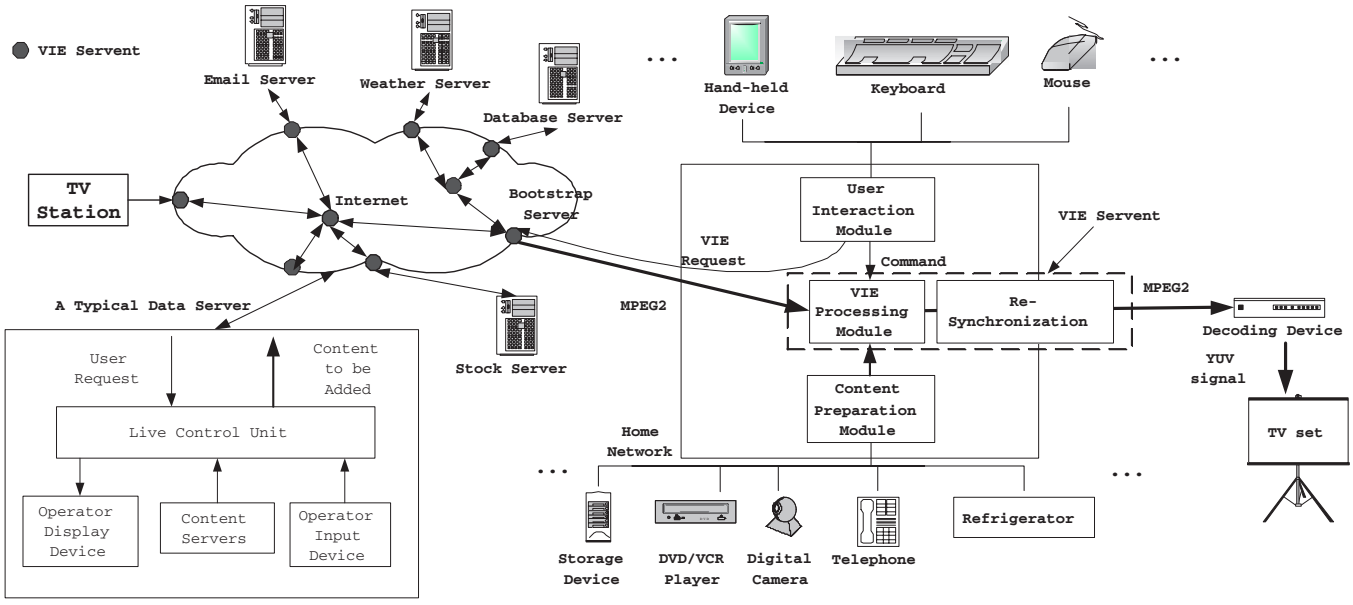


**Fig. 3.** HDControl interactive Internet TV architecture

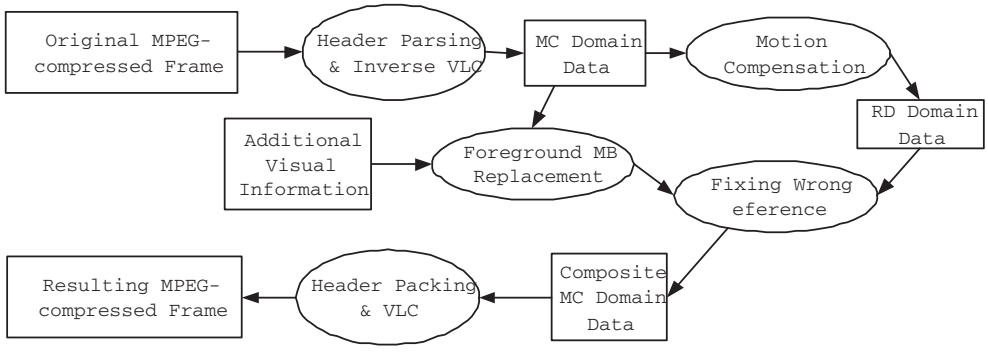


**Fig. 4.** Visual information embedding process

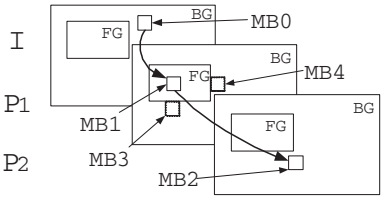


**Fig. 5.** Wrong reference problem

timestamps accordingly to ensure proper clock recovery at the receiver's decoding device (Sect. 4).

### *3.1. VIE and wrong reference problem*

The VIE process aims to embed visual information into an MPEG2 stream directly at the compressed domain, and Fig. 4 shows the primary operations in the VIE process. To achieve VIE, we take the original MPEG compressed frame, parse it through the macroblock headers, and perform inverse variable length coding (VLC) to reach the so-called *motion compensation (MC)* domain, which contains macroblock headers with information such as motion vectors and prediction errors in quantized DCT format. Then, motion compensation is done to get the *reconstructed DCT (RD)* domain data, which contain the DCT-formatted values for each image block. At this point we replace those macroblocks in the *foreground area* (the embedded window) with macroblocks of the visual information to be embedded, and the resulting data are encoded back to an MPEG2 frame.

However, the embedding leads to a *wrong reference* problem because some of the changed macroblocks are used as prediction references for future macroblocks. More specifically, let us consider the chain of $I \rightarrow P_1 \rightarrow P_2$ frames in Fig. 5. The foreground area is the smaller rectangle (FG), and the background frame area is the large rectangle (BG). The small squares represent macroblocks (MB). Let us assume that MB2 in frame $P_2$ uses the data of MB1 in frame $P_1$ as a reference for prediction. Due to VIE, the data in MB1 are changed by the content of the embedded objects. If MB2 uses these new data for its prediction and adds the original prediction error, the final result is obviously wrong. To make matters worse, this wrongly decoded macroblock may then be used as a reference itself for other macroblocks in later frames, causing the error to propagate all the way down the motion prediction link until the next I frame appears. To fix MB2's MC data, we need to know MB2's RD domain data and therefore MB1's. Also note that MB1 itself may rely on the data from another macroblock (MB0 in Fig. 5) for prediction reference. Therefore, to know MB1's value, MB0 will

also need to be decoded to the RD domain. In the worst case, for a GOP pattern of IBBPBBPBBPBBPBB, the data of one macroblock in the first I frame may be referenced four times to derive the content of macroblocks in all four of the following P frames and many other macroblocks in B frames. For a maximum search distance of 16 macroblocks used by the encoder to search the optimal prediction block, this means the prediction link may stride $4 \times 16 = 64$ macroblocks across the background area. Therefore, potentially all the macroblocks in the I and P frames need to be decoded to the RD domain in case future macroblocks need it. In the following discussion, we will call macroblocks like MB0 or MB1 *d-MBs* since their *data* are needed to be decoded to the RD domain for future reference by other macroblocks. Similarly, we will call macroblocks like MB2 *c-MBs* as their reference blocks are wrong and so their MC data have been *changed*.

### 3.2. Previous work on information embedding

Many MPEG compressed domain algorithms have been developed for manipulating video frames, among which [7, 12, 10] provide a good starting point for our work.

In [7], Chang and Messerschmitt thoroughly defined how to do motion compensation and motion reestimation in the DCT compressed domain and proposed the *inference principle* that achieves great computation savings ($10\%$ to $30\%$ speedup according to [7]) in calculating new MC domain data. However, their video manipulation is still a costly process overall, and the motion compensation feature for getting the RD domain data for reference becomes the bottleneck. Though speedup compared to the spatial domain approach has been achieved, real-time processing is still not readily feasible. For example, for HDTV stream with a resolution of $1920 \times 1088$, the decoding step alone is not feasible for real-time processing on general-purpose single-processor PCs.

Based on [7], Meng and Chang [12] proposed a faster solution to the problem of visible watermark embedding. The situation is different in that the watermark block is simply *added* to the original background data:

$$E_{ij} = E_{ij} + W_{ij}, \tag{1}$$

$$E_{ij} = E_{ij} - W_{reference} + W_{ij}. \tag{2}$$

For intracoded macroblocks, Eq. 1 is used to add the watermark value $W_{ij}$ to the image value $E_{ij}$ for the $ij$-th block. For intercoded macroblocks, Eq. 2 is used because we need to first subtract the watermark added to the reference macroblock $W_{reference}$ from the original prediction error $E_{ij}$. This way, the motion compensation work for reference macroblocks is eliminated since we only need to know how they have changed and can adjust the prediction errors based on the changes calculated using only the watermark embedded.

Subsequent to this, Nang et al. [10] pointed out that [12]'s approach may not work: if the embedded watermark or captions are too *strong* (luminance values are too large), Eq. 1 will make $E_{ij}$ overflow the [16,235] bound for a pixel's luminance value. In this case, the result will be cut off, but then the changes made to the prediction error would also depend on the original reference macroblock's data. Therefore, to truly prevent overflow, the maximum luminance value of the reference macroblock is needed again and the savings of [12] are no longer available. The authors of [10] avoided this problem heuristically by using the average luminance of the background block to estimate the caption threshold, which is the DC coefficient in the DCT domain and can be easily acquired. The potential problem is that, when the maximum pixel value differs significantly from the average, overflow will still occur, but the authors claim that the resulting image is of acceptable quality.

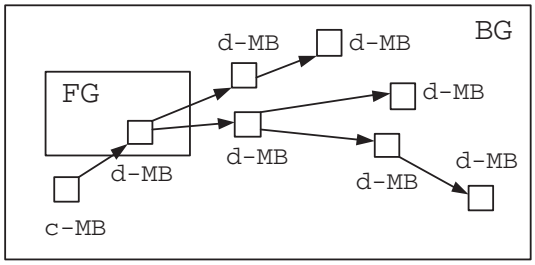


**Fig. 6.** Typical inverse motion prediction link discovered by backtracking

In summary, for general-purpose video/image/text embedding, we must have the original macroblock value of both the reference macroblocks and the dependent macroblocks in the RD domain via motion compensation and so cannot avoid it as in [10,12]. Therefore, we have to face the computation-intensive motion compensation operation and find a new solution to minimize the amount of computation involved in motion compensation.

### 3.3. Solution for efficient motion compensation

Our key observation is that previous solutions are doing much more work than the minimum decoding necessary for rendering correct results. Originally, the RD domain data served two goals: (1) getting the correct values of the d-MBs and c-MBs and (2) calculating new MC domain data through motion estimation. Since Chang et al. [7] simplified the motion reestimation part by only examining the edge macroblocks from the background frame surrounding the foreground area, the second goal can be satisfied as long as we always decode the edge macroblocks, which we call the *gold edge*. For the first goal, in cases where there are only a few c-MBs, the corresponding number of d-MBs will also be small, and the total number of macroblocks needed for reconstruction is much less than the total number of macroblocks in I and P frames. Our experimental results have also confirmed that only those macroblocks surrounding the foreground window are affected by the VIE process and most other macroblocks are not used at all. The only reason that Chang et al.'s solution [7] makes the efforts to completely reconstruct all reference frames is that the future motion prediction pattern is not known in advance. For example, in Fig. 5, when we decode frame P1, we do not know that only a few d-MBs like MB1 will be used as reference for future macroblocks like MB2 in frame P2.

The insight behind our approach is that, for most distributed interactive video applications, various delays exist, such as queueing delay inside the network, buffering/synchronization delay at the receiver side, and delays due

to jitter control and traffic policing. Therefore, for the VIE service case, we expect users to accept the service even if it will introduce a slightly larger delay, or a larger response time for interactive applications, as long as the extra content is embedded in an impromptu manner.

Under this assumption, we can simulate *predicting the future* by *buffering the past*. That is, we decode each frame to the MC domain and buffer the motion vectors and quantized DCT coefficients. After we have gone through the whole GOP, all the c-MBs can be identified by testing whether a macroblock's motion vectors are pointing to somewhere inside the foreground area. Similarly, d-MBs can be identified if some future d-MBs or c-MBs are using it as a prediction reference. Formally, we can define a **backtracking** process as follows:[3]

> From the last B frame to the first I frame in a GOP, for each macroblock $MB\_current$ in the frame, if it uses reference macroblock(s) $MB\_reference$ in the foreground area, then we mark $MB\_current$ as a c-MB and $MB\_reference$ as d-MBs. Also, for every d-MB, all the macroblocks it refers to will also be marked as d-MBs. In addition, all macroblocks on the gold edge are by default marked as d-MBs.

This way, all the *active* prediction links that are relevant to the motion compensation will be found out. Typically these links take the format of *c-MB* $\rightarrow$ *d-MBs* $\rightarrow$ ... *dMBs*, such as the link $MB2 \rightarrow MB1 \rightarrow MB0$ in Fig. 5. Figure 6 shows what a typical inverse prediction link looks like. Note that at each step the number of macroblocks may double or quadruple since the prediction does not follow regular $16 \times 16$ macroblock boundaries. Once the c-MBs and d-MBs are marked out, we can resume the decoding and embedding process from the I frame of this GOP again. At this time we have the MC domain data, and we can continue the VIE process in this way: we perform motion compensation only for those c-MBs and d-MBs that are marked to get their RD domain data, and we perform motion estimation only for c-MBs to get their new motion modes and prediction errors. For other macroblocks, we do nothing, and their MC domain data will be directly reused in a later reencoding phase. We also want to point out that the work done in *backtracking* is always needed within the motion compensation process for any solutions so that it does not incur any extra cost. Rather, instead of backtracking motion vectors and performing motion compensation for each single frame, we defer the backtracking to the end of a whole GOP of frames and use the knowledge acquired to cleverly do motion compensation. Great savings can be expected since only c-MBs and d-MBs need motion compensation, which is typically much less than the total number of macroblocks in I and P frames. This VIE solution is especially useful for timely delivery service for multiple multimedia streams from different sources to a single TV receiver. In addition, the idea of identifying the minimum set of macroblocks for motion compensation through backtracking can also be applied to many other video manipulation operations such as video clipping and video tiling.

[3] For a detailed algorithm, please refer to [14].

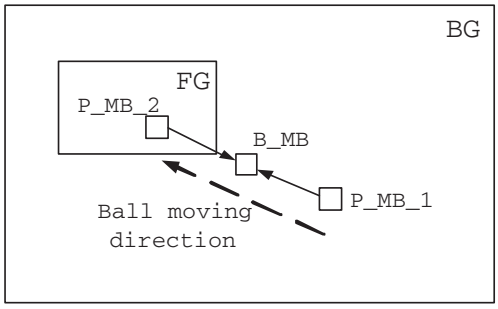


**Fig. 7.** Bidirectional prediction

### *3.4. Optimizations*

Since the VIE process is quite complicated and involves many operations, there are a lot of places where we can make tradeoffs and optimize toward a certain QoS metric. The following are three examples:

**Bidirection to unidirection prediction:** For a c-MB that is bidirectionally predicted, an interesting phenomenon is that it is very likely that only one of its prediction links will point to the foreground area while the others will not. The corresponding physical meaning can be explained this way. Let us assume a jumping ball in a video moving directly toward the foreground area, as in Fig. 7, which shows the position of three macroblocks that describe this scenario in three neighboring P-B-P frames. When encoding a macroblock for the B frame, the encoder may choose to predict it bidirectionally and use the average of the two macroblocks (P_MB_1 and P_MB_2) in the two P frames. Since P_MB_2 is located inside the foreground area, B_MB will be marked as c-MB and the two P frame macroblocks will be marked as d-MBs. Since B_MB is not very different from the two P macroblocks, we can exploit this phenomenon to reduce processing time by discarding the broken prediction. That is, we change the macroblock in the B frame to be unidirectionally predicted using only the one macroblock that is not in the foreground area. The same prediction error can be used since all three macroblocks are basically of the same content, and then the only operation we will need to do will be to change the motion compensation mode from bidirectional to unidirectional and delete one motion vector. In this example, we will only need to change the B_MB to be unidirectionally predicted from P_MB_1, and all three of these macroblocks will not be marked as c-MB or d-MB at all. Although the resulting picture will have a slightly lower visual quality, the experimental results show that it is not obvious to the naked eye and the saving in processing power justifies this cost.

**Sensitive area:** Sometimes the foreground window only occupies a small portion of the background frame, and thus many macroblocks from the background stream may never be affected by the VIE operation. Therefore, we can define a *sensitive area* for each frame for the back tracking process. A *d-sensitive area* specifies the area that may contain d-MBs, and a *c-sensitive area* specifies the area that may contain c-MBs. The benefit of defining sensitive areas comes from saving the decoding work needed for macroblocks outside the sensitive area. If a whole *slice* of macroblocks is *insensitive*, we can copy it from the input stream directly to the output stream

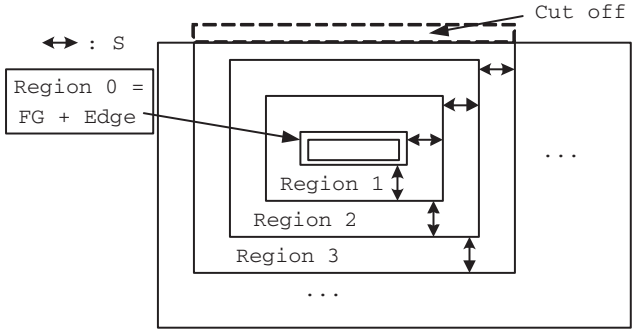


**Fig. 8.** Sensitive regions: candidate macroblocks for backtracking

**Table 1.** Sensitive regions for each frames

| Frame | I | B | B | P | B | B | P | B | B | P | B | B | P | B | B |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| c | – | 1 | 1 | – | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| d | 5 | – | – | 4 | – | – | 3 | – | – | 2 | – | – | 0 | – | – |
| Final | 5 | 1 | 1 | 4 | 1 | 1 | 3 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 |

without even decoding it to the MC domain. In case part of a slice still lies inside the sensitive region, we will still need to decode these slices to the MC domain, but macroblocks outside the sensitive area do not need to be checked for c-MBs or d-MBs in the backtracking process.

For convenience, we define $S$ to be the maximum search distance (in units of macroblocks) used by the encoder during motion estimation (a typical value is 16 macroblocks) and define *Region* $i$ ($i \geq 0$) as follows: Region 0 is the foreground window plus the "gold edge" – the one circle of macroblocks in the background frame surrounding the foreground window. Region $i$ is acquired by enlarging Region ($i$-1) in all four directions by $S$ macroblocks. If the enlarged region exceeds the frame boundary, then the surplus part will be cut off. This is shown in Fig. 8. For every intercoded frame, the c-MBs will only be referring to macroblocks inside Region 0 (the bidirectional case can be eliminated by transforming it to unidirectional prediction), and so the c-sensitive area for every frame is Region 1. The d-sensitive area changes as the backtracking process progresses – the further back we go, the wider the area of macroblocks that may be touched by the prediction link. For the last P frame, the d-MBs are located inside the foreground area plus the gold edge (as needed for motion reestimation), which is Region 0. Therefore, the final sensitive area for the last P frame is the superset of its c- and d-sensitive areas, which is Region 1. All these macroblocks may need macroblocks in the penultimate P frame for reference, and thus those reference macroblocks (candidate d-MBs) are at most $S$ distance away from the gold edge. Therefore, the d-sensitive area for the penultimate P frame is $S$ macroblocks larger than Region 1, which is shown as Region 2 in Fig. 8. Based on a similar deduction, we get the final sensitive area for every frame as shown in Table 1. The numbers represent region indices (– means not applicable), and the final sensitive area of each frame is the super set of that frame's c- and d-sensitive areas.

We will see that when the foreground window is located on the boundary of the background frame, which is the normal case, many slices and macroblocks will be in the insensitive area and require little decoding/encoding operation.

**Shortening the delay:** The longer delay and/or response time caused by the buffering of a GOP of frames is the major cost of our approach. Since $Delay = GOP\_size/frame\_rate$, for a video clip at 30 fps, a GOP of 15 frames means 0.5-s delay. As stated above, this delay is used to wait for knowledge about *future* reference patterns so that we do not do unnecessary motion compensation. However, we can reduce the delay in two ways. First, we can select a shorter GOP size at the encoding time or insert a transcoding proxy to shorten the GOP size. For a GOP size of six frames with 30 fps, the delay will be only 200 ms. If the frame rate is higher, the delay will also be shorter. Of course, a shorter GOP or a higher frame rate means a larger bandwidth requirement for the same video quality or a reduced video quality under the same bandwidth. Therefore, the change of GOP size is rather an engineering choice that balances between video quality, bandwidth requirement, and processing delay. Second, since the last P frames have a relatively small sensitive area, we can start the backtracking process earlier, if necessary, by assuming all the macroblocks in that sensitive area are d-MBs. For example, after we have decoded the penultimate P frame, if we assume all macroblocks in its d-sensitive area are d-MBs, then we can start the backtracking from this point immediately instead of waiting for the remaining four B frames and one P frame to come. This $5/15 = 33\%$ speedup comes at the price of having more d-MBs and thus more motion compensation decoding. The earlier we start the backtracking, the more d-MBs we decode without using them. If we push this to the extreme, then all the macroblocks in the I frame are assumed to be d-MBs, and we are back to the same situation as in [7]: no delay and hence no saving in decoding. This provides another way of balancing the delay and processing time. The more processing power we have, the less delay we may achieve.

## 4. Resynchronization

As mentioned in the introduction, another major challenge to our HDControl architecture is the synchronization problem caused by video editing operations such as VIE and low passing (discarding AC coefficients above a threshold). In this section, we will present some background information on MPEG2 transport stream's synchronization mechanism and then describe in detail the synchronization problem and our solutions.

### *4.1. Synchronization points in MPEG2 transport stream*

Figure 9 shows how MPEG2 transport streams manage to maintain synchronization between the sender, which encodes the stream, and the receiver, which decodes it. As the video and audio data units are packetized, their target *decoding timestamp* (DTS) and *presentation timestamp* (PTS) are determined based on the current sender clock and inserted into the packet headers. For video streams, the access unit is a frame, and both the DTS and PTS are given only for the first bit of each frame after its picture header. These timestamps are later used by the decoder to control the timing at which it starts to do

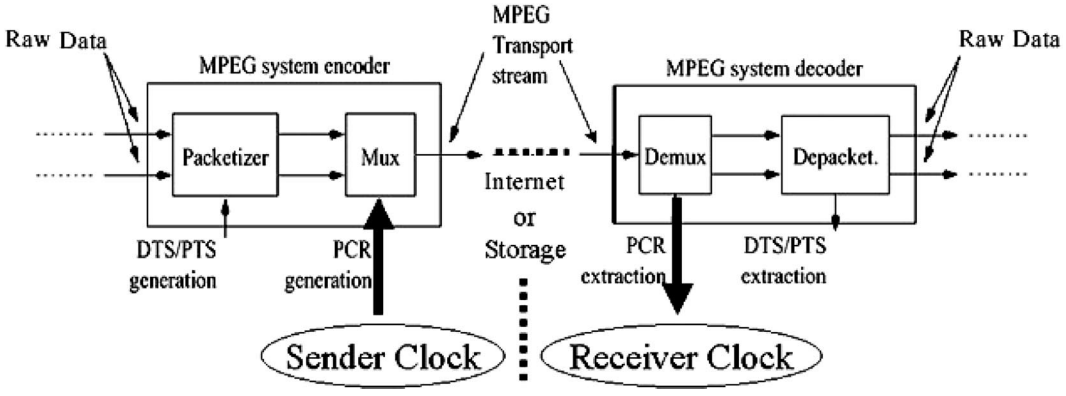


**Fig. 9.** Synchronization between the encoder and decoder

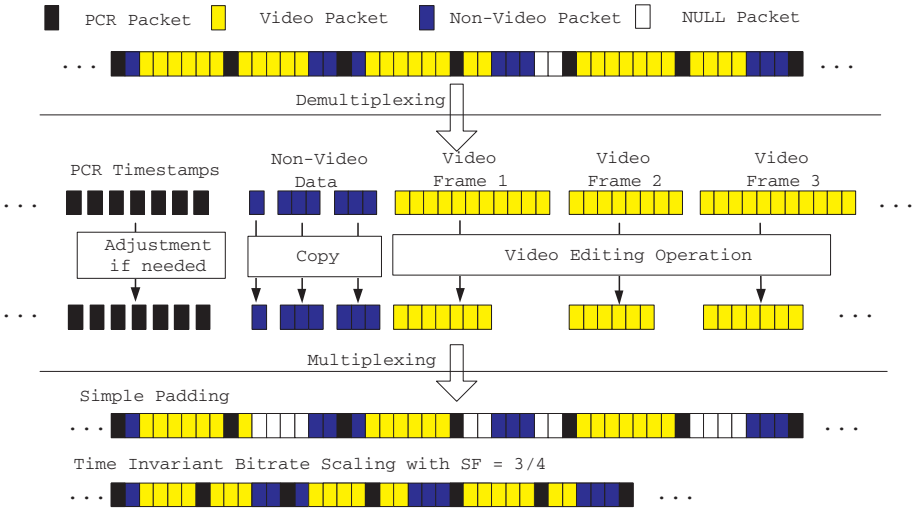


**Fig. 10.** Demultiplexing, video editing, and multiplexing at intermediate video editing servents

decoding and presentation. After that, as all of these packets are multiplexed together, the final stream is timestamped with synchronization points, called *program clock reference* (PCR), that are acquired by periodically sampling the encoder clock. This resulting transport layer stream is then sent over the network to the receiver or stored in storage devices for the decoder to read in the future. As long as the delay that the whole stream experiences remains constant from the receiver's point of view, the receiver should be able to reconstruct the sender's clock that was used when the stream was encoded. The accuracy and stability of this recovered clock is very important since the decoder will try to match the PTS and DTS against this clock to guide its decoding and displaying activities.

Given the general idea in timing, we now introduce how the transport layer syntax works, as shown in Fig. 10. All substreams of raw data (video, audio, data, and timestamps) are segmented into small packets of constant size (188 bytes), and the Packet ID (PID) field in the 4-byte header of each packet tells which substream that packet belongs to. The PCR packets are placed at *constant intervals*, and they form a runtime axis along which all other packets are positioned at the target time point. On this time axis, each 188-byte packet occupies one time slot, and the exact timestamp of each packet/slot could be interpolated using neighboring PCR packets. Data packets are read into the decoder buffer at a constant rate, and this rate can be calculated by dividing the number of bits between any two consecutive PCR packets by the time difference between their timestamps. In other words, if the number of packets between any two PCR packets remains constant, then the difference between their timestamps should also be constant. Ideally, packets are read into the decoder at the constant bitrate, and whenever a new PCR packet arrives, its timestamp should match exactly with the receiver clock, which confirms that the decoder has so far successfully reconstructed the same clock as the encoder's. However, since PCR packets may have experienced jitter in network transmission or storage device access before they arrive at the receiver, we cannot simply set the receiver's local clock to be the same as the timestamp carried by the next incoming PCR regardless of when it arrives. To smooth out the jitter and maintain a stable clock with a limited buffer size at the receiver, generally the receiver will resort to some smoothing technique like the phase-locked loop (PLL) [9] to generate a stable clock from the jittered PCR packets. The PLL technique is a feedback loop that uses an external signal (the incoming PCR packets in our case) to tune a local signal source (generated by a local oscillator in our case) to generate a relatively more stable result signal (the receiver's reconstructed local clock in our case). So long as the timing relation between PCR packets is correct, the jitter can be smoothed out with a PLL.

### *4.2. Resynchronization problem at video editing servents*

As shown in Fig. 10, when the incoming stream arrives, the video editing servents demultiplex it into PCR timestamps, nonvideo data, and video frames, exercise video editing operations on the video frames, and then multiplex the resulting frames with other data to form the output stream. The challenge comes from the fact that the frame size will be changed by the editing operations. For example, after a VIE operation the frame size may be larger if the embedding content is of lower compression ratio; after a low passing operation, some high-frequency AC coefficients will be discarded, and so the frame size will decrease. This change in frame size means the number of 188-byte packets used to carry data for each frame will also be changed, and if we naively pack the resulting packets together, the constant spacing of PCR packets will be violated. Also, the relative position of each video/audio frame on the timeline will be changed, which will also affect the decoding process. Therefore, the key problem is how to correctly and efficiently do the multiplexing after the video editing operations at the intermediate video editing servent so that the synchronization between the original encoder and decoder is maintained. We call this problem the *resynchronization problem*. Formally, the resynchronization problem may be described as follows:

> Given the incoming MPEG2 transport stream, each packet is represented by a tuple (*sequence_no, frame_number, type*), where $sequence\_no \geq 0$ is the packet sequence number, $frame\_number \geq 0$ is the sequence number of the video frame this packet belongs to, and $type \in [PCR, NONVIDEO, VIDEO, NULL]$. As a result of the video editing operation, the number of packets with *type = VIDEO* may increase or decrease, so the original *sequence_no* will not be correct. The goal is to assign a new sequence number to each packet (original nonvideo packets and new packets from output of video editing operations), so that the resulting stream's PCR packets are spaced with constant distance and non-PCR packets are placed at the right time point between the PCR packets.

### *4.3. Our solutions*

To solve the resynchronization problem, an immediate solution would be to do the same kind of clock reconstruction as the decoder does at the video editing servents and then regenerate new PCR packets to reflect the changes caused by the editing operations. The smoothing mechanisms like PLL can help, but they are implemented in hardware circuits containing a voltage-controlled oscillator that generates high-frequency signals to be tuned with the incoming PCR timestamps. This is hard, if not impossible, to do in software on computers without real-time support in hardware. Hence this hardware solution is not a viable solution as our goal is to find a pure software solution that would enable us to distribute video editing servents across a network to any point along the streaming path. Another goal for us is to achieve a cheap and efficient solution that could be easily implemented and carried out by any computer with modest CPU and memory resources available.

The key idea behind our solution comes from the observation that the DTS and PTS are only associated with the beginning packet of each frame. Consequently, as long as we manage to fix that point to the correct position on the time axis, the decoder should work fine even if the remaining bits of that frame following the starting point are stretched shorter or longer.

#### 4.3.1. Simple padding

Following the discussion above, we have designed a simple solution that works for bitrate-reducing video editing operations, as shown in Fig. 10. We do not change the timestamp and the position of any PCR packet along the time axis within the stream, and we also preserve the position of the frame header and the beginning bit of every frame. What is changed here is the size of each frame in terms of the number of bits, and we just pack the resulting bits of a frame closely following the picture header. Since each frame takes fewer packets to carry, yet the frame headers are still positioned at their original time points, we can imagine that there would be some *white space* left between the last bit of one frame and the first bit of the header of the next frame. Actually, the capacity of this space is the same as the reduction in the number of bits used to encode this frame as a result of the video editing operations, and we can simply *pad* this space with *empty* (NULL) packets.

This solution is very simple to understand and implement, and it preserves synchronization since we only need to pack the filtered bits of each frame continuously after the picture header and then insert NULL packets until the header of the next frame. No new timestamps need to be generated in real time, and the bitrate remains stable at the original rate. However, the solution inevitably has two drawbacks. First, it can only handle bitrate-reducing operations. We only try to fix the header of each frame to its original position on the time axis, which means the changed frames should not occupy more bits larger than the distance between the current frame header and the next. This property does not always hold since some video editing operations like VIE and watermarking may increase the frame size. Second, the saved bits are padded with NULL packets to maintain the original constant bitrate and the starting point of each frame, and this runs against the initial goal of bitrate-reduction operations like low pass filtering and color/frame dropping. The resulting stream contains the same number of packets as the original one. The only difference is that the number of bits representing each frame has been shrunk, yet this saving is spent immediately by padding NULL packets at the end of each frame. Though we can compress these extra NULL packets with a special protocol between the encoder and decoder, that would no longer conform with the standard MPEG2 transport syntax and would compromise interoperability.

#### 4.3.2. Time-invariant bitrate scaling

To ultimately solve the synchronization problem, a more general algorithm has been designed. The key insight behind this algorithm is that we can change the bitrate to another *constant* value while preserving the PCR timestamps by changing the number of packets between any PCR pair to another constant value. The ratio between the original PCR packet distance

and the resulting PCR packet distance is called the *scaling factor* (*SF*). This way, we can scale the PCR packets' distance and achieve a fixed new bitrate, as if the time axis is scaled looser or tighter to carry more or less packets. Yet we do not need to regenerate new PCR timestamps that rely on hardware real-time support. All nonvideo stream packets can be simply mapped to the new position on the scaled output time axis that corresponds to the same time point as on the original input time axis. If no exact mapping is available because the packets are aligned at units of 188 bytes, we could simply use the nearest time point on the new time axis without introducing any serious problems. For video stream, the same kind of picture header fixing and frame data packing are conducted as in the first solution but in a scaled way. In the example given in Fig. 10, where SF is $3/4$, there is one PCR packet every six packets instead of eight packets as in the original stream, and no NULL packet is needed. Note that *simple padding* is the special case where *SF* is 1.0. Formally, the time-invariant bitrate scaling algorithm goes as follows:

```
For each packet p do {
  let q be p's previous packet;
  if (p.type is not Video ) {
    p.sequence_no = floor(p.sequence_no * SF);
    if (p.sequence_no < q.sequence_no)
      p.sequence_no = q.sequence_no + 1;
  } else {
    let F_after be the frame after editing that p
    belongs to;
    let F_before be the same frame before editing;
    if (p is the first packet of F_after) {
      let q be the starting packet of F_before
      p.sequence_no = q.sequence_no * SF;
      if (p.sequence_no < q.sequence_no)
        p.sequence_no = q.sequence_no + 1;
    } else {
      let s be p's previous packet in F_after
      p.sequence_no = s.sequence_no+1;
      if (p is the last packet in F_after ) {
         let t be F_before's last packet
         for (i=p.sequence_no+1;i<=t.sequence_no*SF;i++)
              insert a NULL packet with sequence_no i
} } } }
```

This algorithm is also very simple to implement. For each nonvideo packet, its distance (in number of packets) from the last PCR packet is multiplied by a scaling factor $SF$, and the result is used to set the distance between this packet and the last PCR packet in the output stream. For video frames, the header containing DTS and PTS is scaled and positioned in the same way, and the remaining bits are closely appended to the header in the result stream.

Now the only problem is how to determine $SF$ for a specific video editing operation. If we shrink the time scale too much and for some frames the editing does not achieve a significant bitrate reduction, then we will not have enough space to squeeze in this frame, which will push the beginning bit of the next frame behind schedule. On the other hand, if we shrink the time axis too little or expand ( $SF > 1$) it too much, then more space will be padded using NULL packets to preserve important time points, leading to a waste of bandwidth. There exists one optimal scale factor $SF_{opt}$ that can balance these two strengths if it fulfills the conditions that (1) every resulting packet can be assigned a valid sequence_no and (2) the number of NULL packets used for padding is kept to a minimum.

However, this optimal scale factor is hard to estimate in advance since different operations with various parameters have quite varying effects on different video clips in terms of bitrate changing. Therefore, in our current implementation, we have adopted a two-stage approach. First, we measure offline the *maximal ratio* ($R$) in number of packets of a stream before and after the video editing operation for typical video streams and operation parameters. Second, when the streaming has started, we monitor online the *actual ratio* ($AR$) of change of the number of packets (excluding NULL packets) between each pair of PCR packets before and after the video editing operation. The scaling factor is calculated using the following formula: $SF_{new} = \alpha * R + (1 - \alpha)*$ (average AR over the last $N$ pairs of PCR packets). Note that $\alpha$ controls how offline experience determines current $SF_{new}$ values, and $N$ is the window size of how many previous PCR pairs will affect the current $SF_{new}$ value. For the decoder to reconstruct a stable clock, $SF$ should not be changed frequently, so we only update $SF$ with $SF_{new}$ when their difference is larger than an empirical threshold.

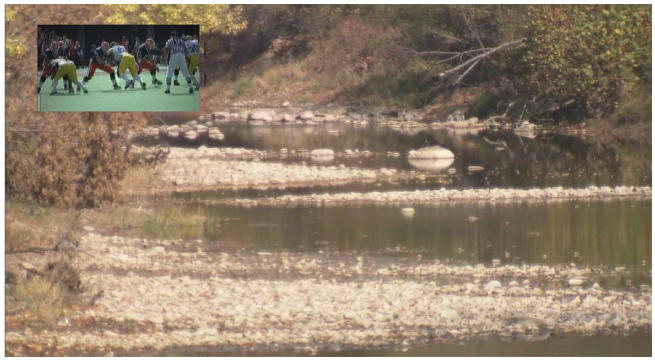

**Fig. 11.** An example screen shot for PiP

## 5. Experimental results

### 5.1. HDTV testbed

Because of the limitation in full deployment of our proposed architecture in the wide area Internet, we have built a testbed for HDTV streaming and editing within our department local area network (LAN). Live high-definition digital TV stream from the satellite or the HD storage device is fed into the server PC, which then encodes the HD video into MPEG2 transport stream and multicasts this stream over the high-speed LAN. Players on the user PCs join this multicast group to receive the HD stream and feed this stream into the HDFocus decoding board [1]. The decoded analog signal is then sent to the wide-screen TV for high-quality display. Video editing gateways (servents) receive this stream in the same way as a normal player and perform various video editing operations on this stream in real time, such as low pass filtering, frame/color dropping, and VIE. There can be multiple editing operations done to the same stream in a chain, and the resulting streams at all stages are available to users through different multicast groups.

**Table 2.** Comparison of number of macroblocks that need motion compensation

| | Football | Stars | Trees |
|---|---|---|---|
| Previous approach | 1360000 | 1360000 | 1360000 |
| Our approach | 139455 | 55509 | 56613 |

### *5.2. VIE results*

We analyze the performance of the proposed VIE approach with respect to aspects:

**Real-time processing:** The VIE service gateway runs on a general-purpose PC with a single Pentium IV 1.4-GHz processor and 512 MB memory. The program is written in Visual C++ and runs on Windows 2000 platform. As expected, we can perform many VIE functions in real time, introducing an extra delay of approximately 0.5 s. After initialization, the processor utilization levels off at 70% when no other applications are running on the gateway machine at the same time. For this experiment, we have chosen the background stream to be a HD quality ($1920 \times 1088$, 30 fps) stream "Trees1.mpg", and the foreground stream to be a standard resolution MPEG video "football_sd.mpg" ($480 \times 256$, 30 fps ). Figure 11 presents a screen shot of the resulting stream. Note that the foreground window can be located anywhere in the frame as long as it follows the $16 \times 16$ macroblock boundaries.

**Computational complexity reduction:** As has been analyzed above and was also pointed out by Chang [7], a very expensive computation done using Chang's approach is the decoding of reference macroblocks from MC domain to RD domain, and our approach consists precisely in solving this problem based on the observation that normally a large portion of these macroblocks is not used at all.

To demonstrate this, we only need to compare the number of macroblocks that are decoded to the RD domain by the two approaches. In the approach in [7], all the macroblocks in all I and P frames should be counted; thus for the "football in trees" example in Fig. 11, the total number of macroblocks decoded to RD domain for 500 frames is $\frac{1920\times1088}{16\times16} \times \frac{5}{15} \times 500 = 1360000$. On the other hand, for our approach the total count depends upon the nature of the background stream since we only work on affected macroblocks. For three different background streams, the total number of macroblocks reconstructed through motion compensation is given in Table 2.

From Table 2 we know that with our approach the most expensive motion compensation operation only needs to be done for less than 10% of the previous approach, and for background streams such as stars and trees this percentage is even smaller. To further explain the saving, we have plotted out the distribution of d-MBs over the whole background area. Specifically, for each MB position over the $\frac{1920}{16} \times \frac{1920}{16} = 120 \times 68$ MB coordinates, we counted how many times that MB is marked as a d-MB for 300 consecutive frames, and the embedded window's coordinate is [6, 4, 36, 20] (the same as illustrated in Fig. 11). As shown in Fig. 12, only MBs near the window edge are likely to be actually decoded by the VIE process.

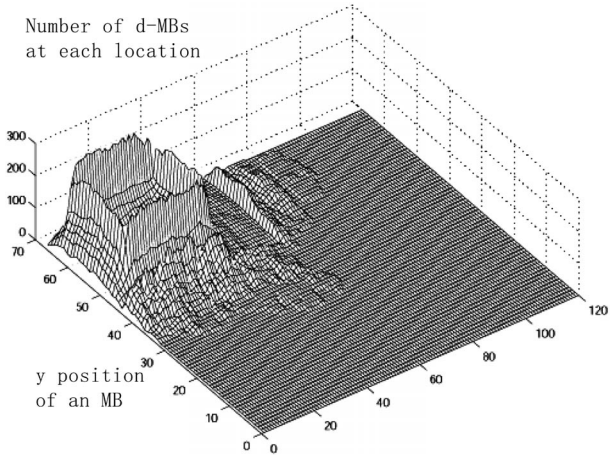


**Fig. 12.** Distribution of d-MBs

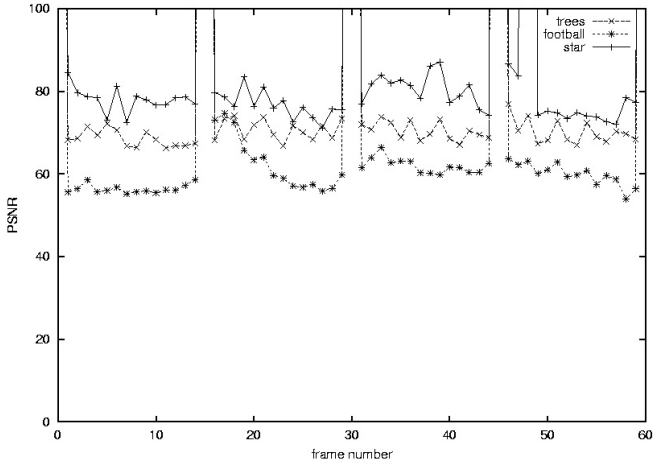


**Fig. 13.** pSNR of background image for 3 video clips

**Resulting video quality:** We will perform the image quality pSNR analysis, as done in other studies such as [7]. The quality degradation primarily comes from DCT quantization, simplified motion estimation, and some of the optimizations we discussed. For the visual quality test, because the foreground video is not changed to opaque overlaying, we only calculate the pSNR (peak signal-to-noise, Eq. 3) values for the background streams:

$$pSNR(X,Y) = 20log_{10}\left[\frac{255}{\sqrt{\frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N}(X_{ij}-Y_{ij})^2}}\right]. \quad (3)$$

Specifically, we use a standard MPEG2 decoder to decode both the original background streams and the resulting streams after embedding the overlay stream to pixel domain and use their difference (excluding the foreground area) as the noise value. From the result shown in Fig. 13 we can see that the I frames are not affected at all, and other frames' pSNR value changes for different video clips: the more motions exist in the video stream, the more noise results from the VIE process.

We have also performed a thorough subjective test. Over 100 viewers have seen the resulting stream of our real-time VIE, and no perceptible quality degradation was observed.

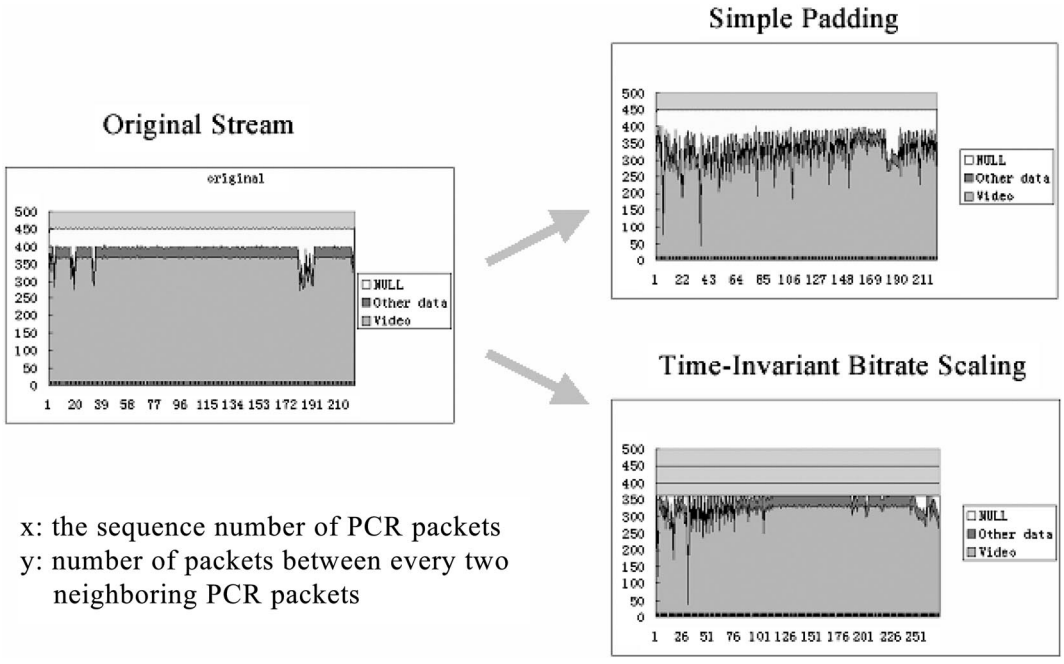


**Fig. 14.** Effect of time-invariant bitrate scaling

*5.3. Resynchronization results*

Figure 14 shows the effect of the time-invariant bitrate scaling approach for low pass operations with an AC coefficient threshold of 5. Each point on the $x$-axis represents a PCR packet, and the $y$-axis shows in three colors how many video packets, NULL packets, or packets for other data streams are present in between this PCR packet and its previous PCR packet. We can see that the distribution of the three areas is kept almost constant for the original frame except for more NULL packets at the end of a frame. For simple padding, the number of video packets varies across different PCR intervals and a lot of extra space is padded with NULL packets, as shown in the upper right panel of the figure. On the other hand, if we do time-invariant bitrate scaling with a scaling factor of $80\%$, then the padding occurs mostly at the end of frames and the stream contains mostly useful data.

Despite the resynchronization process, our scaling approach may still introduce a skewed starting time for some frames. The reason is that after the video editing operation, each frame shrinks to a size mostly less than $80\%$ of its original size. If we mask out all other packets, we can see that in the video stream frames are packed closely one after another. If one frame takes more space than its share, then the next frame may be pushed behind its time point, but this skew will be compensated shortly by another frame with a stronger shrinking effect. As we said before, this kind of small jitter around the exact time point on the scaled time axis is acceptable, and it is the change in the bitrate at which the decoder reads in the data that fundamentally enables our algorithm to solve the problem.

Another experiment shows how to determine the scaling factor $SF$ for a particular kind of video editing operation. Three kinds of operations are tested: low pass filtering (LP) with threshold of 10 and 5 and VIE for picture-in-picture (PiP). The original stream is a HD stream "stars1.mpg" with bitrate 18 Mbps in MPEG2 transport layer format. The embedded frame used for picture-in-picture is a scaled-down version of another HD stream "football1.mpg". Since the content of this stream is more condensed than the background stream (i.e., more DCT coefficients are used to describe each block), it is expected that the bitrate will increase after this VIE operation. The final statistics are shown in Table 3. Experimental results have shown that with the suggested scaling factor based on real-world statistics, our time-invariant bitrate scaling algorithm could successfully solve the synchronization problem.

**Table 3.** Proving for the scaling factor

| | LP (10) | LP (5) | PiP |
|---|---|---|---|
| Original BR (Mbps) | 18.0 | 18.0 | 18.0 |
| Average resulting BR (Mbps) | 15.45 | 13.71 | 19.04 |
| Average relative change | 0.86 | 0.76 | 1.06 |
| Suggested $SF$ | 0.90 | 0.80 | 1.10 |

## 6. Conclusion and future work

Interactive Internet TV that features open standard and service model, high-quality video, dynamic software real-time editing, and great user customization is a very promising area, and in this paper we have presented the HDControl architecture and the corresponding service model. We have also presented two key algorithms that are essential to any real-time systems that work on MPEG2 format content. The experimental results on our local testbed have validated the efficiency and feasibility of our approach.

## References


1. HDFocus: High Definition MPEG Decoding Board. http://www.visualcircuits.com/prod_serv/HD.cfm
2. Gigabit Ethernet (2001) The IEEE international symposium on circuits and systems (ISCAS 2001), tutorial guide, location, day month 2001, pp 9.4.1–9.4.16
3. ISO/IEC 13818 13818 (1994) Generic coding of moving pictures and associated audio information. ISO
4. Amir E, McCanne S, Katz RH (1998) An active service framework and its application to real-time multimedia transcoding. In: Proceedings of SIGCOMM 1998, Vancouver, British Columbia, August 1998, pp 178–189
5. Bezzan C (1990) High definition TV: its history and perspective. Proceedings of the SBT/IEEE international telecommunications symposium (ITS '90), Rio de Janeiro, September 1990
6. Bhagavath VK (1999) Emerging high-speed xDSL access services: architectures, issues, insights, and implications IEEE Commun Mag 37(11):106–114
7. Chang SF, Messerschmitt DG (1995) Manipulation and compositing of MC-DCT compressed video. IEEE J Select Areas Commun 13(1):1–11
8. Dutta-Roy A (2001) An overview of cable modem technology and market perspectives. IEEE Commun Mag 39(6):81–88
9. Holborow CE (1994) Simulation of phase-locked loop for processing jittered PCRs. ISO/IEC JTC1/SC29/WG11, MPEG94/071, March 1994
10. Nang SHJ, Kwon O (2000) Caption processing for MPEG video in MC-DCT compressed domain. In: Proceedings of the 8th ACM Multimedia, Los Angeles, October 2000
11. Zhang LJ, Chung JY, Liu LK, Lipscomb JS, Zhou Q (2002) An integrated live interactive content insertion system for digital TV commerce. In: Proceedings of the international symposium on multimedia software engineering, Newport Beach, CA, December 2002
12. Meng J, Chang SF (1998) Embedding visible watermark in compressed video stream. In: Proceedings of the 1998 international conference on image processing (ICIP'98), Chicago, October 1998
13. Spragins J (1996) Fast Ethernet: dawn of a new network. IEEE Netw 10(2):4
14. Yu B, Nahrstedt K (2002) A compressed-domain visual information embedding algorithm for MPEG2 HDTV streams. In: Proceedings of the international conference on multimedia and expo, Newport Beach, CA December, 2002