AN ABSTRACT OF THE THESIS OF

Reshma Dixit for the degree of Master of Science in Electrical and Computer Engineering presented on April 8, 2004.
Title: Low-Power Video Decoding with Dynamic Voltage Scaling.

Abstract approved:

_____

Ben Lee

This thesis investigates Dynamic Voltage Scaling (DVS) techniques to lower power consumption in video decoding. A DVS scheme called the Frame-data Computation Aware (FDCA) method has been presented. This method is adaptable not only to stored video applications but also to real-time video scenarios. Unlike DVS schemes for video decoding proposed earlier, the FDCA scheme does not require any preprocessing mechanisms. Results from simulations performed using the scheme are presented and compared with prior existing DVS schemes. The results indicate that the FDCA method provides power saving of up to an average of about 68%.

Low-Power Video Decoding with Dynamic Voltage Scaling

by

Reshma Dixit

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented April 8, 2004
Commencement June 2004

Master of Science thesis of Reshma Dixit presented on April 8, 2004.

APPROVED:

Major Professor, representing Electrical and Computer Engineering

Associate Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Reshma Dixit, Author

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation and gratitude to Prof. Ben Lee for providing valuable guidance, advice, suggestions, and comments during my time here at Oregon State University. I would also like to sincerely thank Prof. Alexander Tenca, Prof. Timothy Budd, and Prof. Keith Levien for taking interest in my thesis topic and for setting aside time for this. I would like to thank Eriko Nurvitadhi for all his input, comments, and suggestions.

I would like to thank my family for their tremendous support all this time.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

LOW-POWER VIDEO DECODING WITH DYNAMIC VOLTAGE SCALING

1. INTRODUCTION

Faster processors, larger network bandwidths and the ubiquitous Web have contributed to a great extent to the growing popularity of multimedia applications, and video applications form a major part of these. With advances in processor and wireless technology, smaller battery-operated mobile devices such as laptops, PDAs and mobile phones are not only prevalent today but are now also capable of running complex applications such as video decoding. However, power consumption in portable devices still remains a principal technological challenge and a major design goal. With these portable devices running computationally intensive and thereby power-greedy video applications, the importance of preserving power becomes increasingly important. While on one hand, the battery life of a portable device can be increased by using improved technology at the source i.e. improved battery technology, on the other hand, battery life can also be increased by reducing power consumption at the sinks i.e. at different units of the system consuming power. The CPU can be a very suitable target for reducing

power consumption, especially since it could potentially account for about 11-28% [1] of the total power consumption of the device.

Dynamic Voltage Scaling (DVS) is a power saving technique that has been researched widely in recent years [3-20, 23-31] and has been used to reduce power consumption at the CPU. This technique may be better referenced as Dynamic Voltage/Frequency Scaling, since the principal idea behind this concept is to dynamically vary the voltage and frequency of the processor by taking advantage of the workload variability on the processor. One of the most classic examples of an application that exhibits a great deal of workload variability, thereby requiring variable processing power, is video decoding and therefore, video decoding applications can particularly take advantage of DVS to preserve power. Various studies have been done [23-31] using DVS specifically on video decoding applications and these methods have taken into consideration the various logical building blocks of a video stream such as a Group of Pictures (GOP) or frames in order to decide when and how to change the voltage/frequency setting of the processor to obtain optimum power savings.

Most of these existing DVS power-saving techniques for video decoding require *a priori* knowledge of the video stream in terms of frame sizes and/or a frame size vs. decoding time relationship to predict the decoding time for an upcoming frame or GOP to thereby change the processor setting accordingly. These methods have to rely on some external preprocessing and data generating

mechanisms to input the parameters required for the algorithm since such information is not available before decoding the stream. Such an approach may not be appropriate for real-time video applications such as video conferencing and broadcasting.

The focus of this thesis is to overcome the above-mentioned difficulty in using DVS algorithms for video applications. A DVS technique called the *Frame-data Computation Aware (FDCA)* scheme has been proposed that extracts useful frame-specific information from the video stream *while* decoding the frames and uses this information to estimate the decoding time in order to change the voltage/frequency setting of the processor. Thus, there is no need to rely on preprocessing mechanisms to generate input for the DVS algorithm, which makes this method adaptable to practical real-time video scenarios also.

The rest of this thesis is organized as follows. Chapter 2 provides background information about DVS and its applicability to video decoding. It also explains the generic MPEG decoding model. Chapter 3 is a literature review section that explains briefly previous work done in DVS on video decoding. Chapter 4 presents an overview of our proposed technique for using DVS on video decoding. Chapter 5 describes the simulation environment and workload streams used in our experiments, and explains the parameters used for analyzing our results. Chapter 6 presents and interprets the results from our simulations. Finally, conclusions and future work suggestions are presented in Chapter 7.

2. BACKGROUND

In this chapter, we provide relevant background information and concepts about Dynamic Voltage Scaling (DVS) and how it is applicable to video decoding. We then go on to explain the generic MPEG decoding model.

2.1 Dynamic Voltage Scaling (DVS)

CMOS circuits follow the equations given below for power dissipation (P) and circuit delay (t) respectively [5, 25],

$$P \propto C_{eff} V_{dd}^2 f_{clk} \quad (1)$$

$$t \propto V_{dd} / (V_{dd} - V_T)^2 \quad (2)$$

where, C_{eff} is the effective switching capacitance, V_{dd} is the supply voltage, f_{clk} is the clock frequency, and V_T is the threshold voltage

From equation (1), it can be seen that reducing the supply voltage will lead to reduction in power consumption. However, from equation (2), it is also evident that reducing the supply voltage increases the circuit delay, t. This in turn, implies that when the supply voltage is decreased, the frequency also has to be decreased.

Dynamic Voltage Scaling (DVS) refers to a technique in which the voltage, and simultaneously the frequency of a processor are varied dynamically, as required, in order to reduce power consumption. Use of DVS inherently implies a trade-off between lowering power consumption of the system and

system performance in terms of speed. Several studies have been done using DVS as a power-saving technique. Variable-voltage processors are now also commercially available e.g. with Intel's XScale®, AMD's PowerNow!™, and Transmeta's LongRun™ technologies.

DVS techniques can be broadly classified into two main categories. The first category applies DVS at operating system level [3-15] with a focus on inter-task voltage scheduling. Pering et al. [7] have presented such algorithms taking into account the global state of the system and setting the voltage/frequency for a task using different workload prediction mechanisms. Lorch and Smith [14] have proposed a method to modify and improve on these DVS algorithms by using a technique in which the speed of task execution is progressively increased. The work presented by Pillai and Shin [13] proposes DVS algorithms specifically suited for real-time embedded operating systems.

Since different types of tasks have different processing requirements, it may not be possible for operating system-level voltage/frequency schedulers to predict the processor workload accurately; especially in a case where an application has a highly fluctuating processing demand. Therefore, it is also necessary for applications themselves to be power-aware. The second category of DVS techniques thus includes methods that use power-aware applications to use DVS. DVS on video decoding falls in this category. Shin et al. [16] in their work have presented such an intra-task voltage scheduling technique, while Pouwelse et

al. [17] have used power-aware applications that convey their processing requirements to a central voltage scheduler. Compiler optimization techniques that can be used for DVS have also been proposed [18, 19].

The general trend in carrying out various tasks on a processor has been to try and complete the tasks at the highest possible execution speed, and therefore the highest available voltage setting, in order to make way for other scheduled tasks in the queue. In many cases, a task may not gain any significant benefit from being carried out at the highest processor frequency. At the same time, such an approach leaves idle periods when the processor does no useful work. Video decoding is one such application in which good perceptual video quality may be maintained as long the video stream is played at the specified frame rate. Also, since it is a soft real-time application, a missed deadline does not lead to catastrophic consequences, although it may cause a degradation of video quality. This coupled with the fact, that processing requirements for video decoding may vary erratically makes it a very suitable application for use with DVS for achieving reduced power consumption.

The power-saving advantage that video decoding can achieve with DVS and related challenges can be better understood with examples illustrated in Figure 2-1 through Figure 2-3. As shown, consider a video clip being played at a frame rate of say, 30 frames per second (fps), on a variable-voltage processor. As discussed above, such a processor therefore will also have a particular voltage

associated with each frequency setting and an increase or decrease in frequency of the processor also implicitly indicates an increase or decrease in the voltage respectively. Thus, the y-axis in these figures represents not only the frequency but also the corresponding voltage associated with that frequency setting. Figure 2-1 and Figure 2-2 use a variable voltage processor that can be set to any desired voltage/frequency setting with a maximum frequency of 120 MHz. Figure 2-1 shows three frames of a video stream, with all three being decoded at the highest frequency of 120 MHz i.e. without using DVS. These three frames have different processing requirements represented by the shaded portions. Once a frame is decoded the processor goes into a shutdown mode until the next frame in the stream. When DVS is used with video decoding, ideally the same scenario would look as shown in Figure 2-2. In this figure, the shaded portions still represent the same workload as in Figure 2-1, however, the processor slack times are completely eliminated by decoding the frame at a voltage/frequency level that would just satisfy the processing requirement for the frame and thus it consumes less power than the scenario pictured in Figure 2-1. It is important to note here that Figure 2-2 illustrates an ideal case in which the accurate decoding time of a frame is "known" prior to actually decoding it so that the required voltage/frequency level can be appropriately set. Not only that, the variable voltage processor can be set to any desired voltage/frequency level.

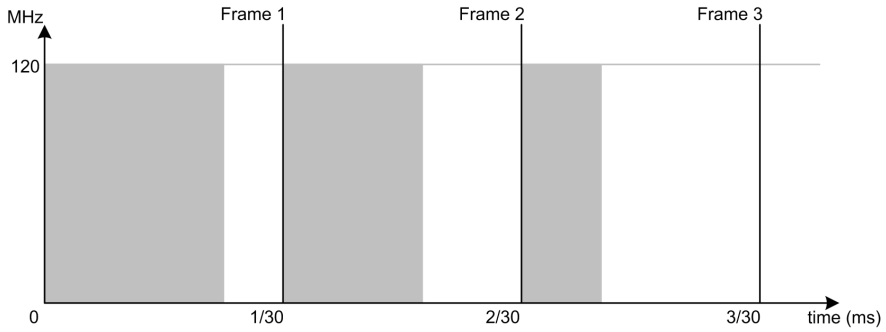


Figure 2-1. Video Decoding with No DVS

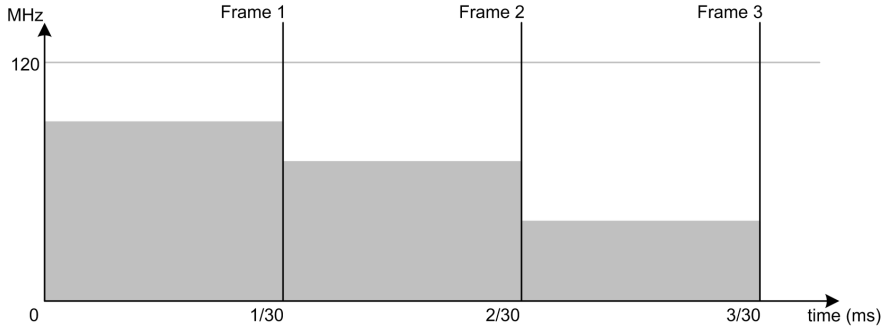


Figure 2-2. DVS on Video Decoding in Ideal Scenario

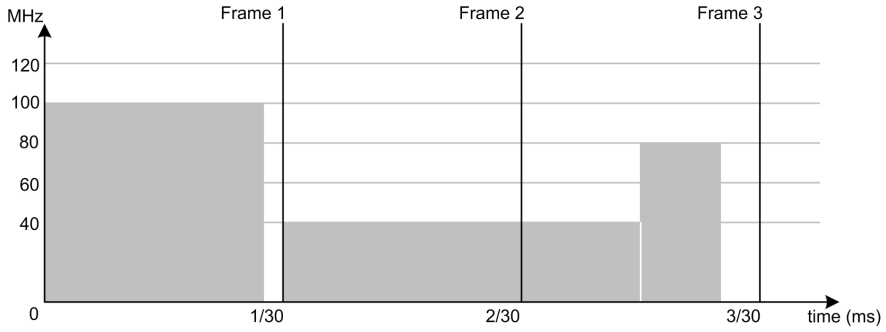


Figure 2-3. DVS on Video Decoding in Practical Scenario

Different video clips have different characteristics and even within a video clip, the processing requirements may vary greatly due to factors such as background or scene changes. One of the major challenges therefore in using DVS on video decoding is prediction of future processor workload. In practice, what usually may happen when using DVS on video decoding is shown in Figure 2-3. The prediction may be fairly accurate (Frame 1), under predicted (Frame 2), or over predicted (Frame 3). Over predictions will cause more power to be consumed while under predictions may cause video quality to be degraded.

The second aspect that must be considered when using DVS on video decoding is the number of voltage/frequency settings available with the processor and that are used in performing DVS [26] as well as the upper and lower voltage/frequency levels of the processor. Although a processor with a voltage/frequency scale range that can be continuously varied is desirable, there is cost associated [2, 4] in the design of such processors. In actuality, variable voltage processors available provide a range of voltage/frequency levels that can be varied in a limited number of discrete steps. This may affect the way in which video decoding applications benefit from DVS. The use of even the most accurate or close to ideal prediction algorithm with a processor that has a very narrow range and few steps of voltage/frequency levels may lead to negligible power saving with DVS because of the fact that the processor is not able to scale well to the accurate prediction and either leaves significant amount of idle periods or is

not able to meet deadline requirements [26]. Figure 2-3, along with showing how the accuracy of the prediction mechanism affects DVS on video decoding, also represents the aspect discussed above.

2.2 Generic MPEG Video Coding and Decoding Model

The MPEG video format is one of the most widely used video formats today not only because it has a very good compression ratio but also because it gives very good video quality and is the format used in such applications as high-quality video DVDs. We take the MPEG format as a representative of the video formats available and present some basic concepts about the generic MPEG video decoding model since it forms an essential part of this thesis.

An MPEG video stream [22] consists of three types of frames: I or intra-coded, P or predictive-coded, and B or bi-directionally-coded predictive frames. The frames are coded with subsampled chrominance components. A Group of Pictures (GOP) is a sequence of I-, P-, and B- frames and typically consists of about 12-15 frames with an I-frame marking the beginning of a GOP.

A frame consists of a series of macroblocks, with each macroblock representing a 16x16 pixel area of a frame. A macroblock consists of a set of six 8x8 blocks of pixels, with four luminance and two chrominance blocks. The structure of a typical MPEG video stream is illustrated in Figure 2-4.

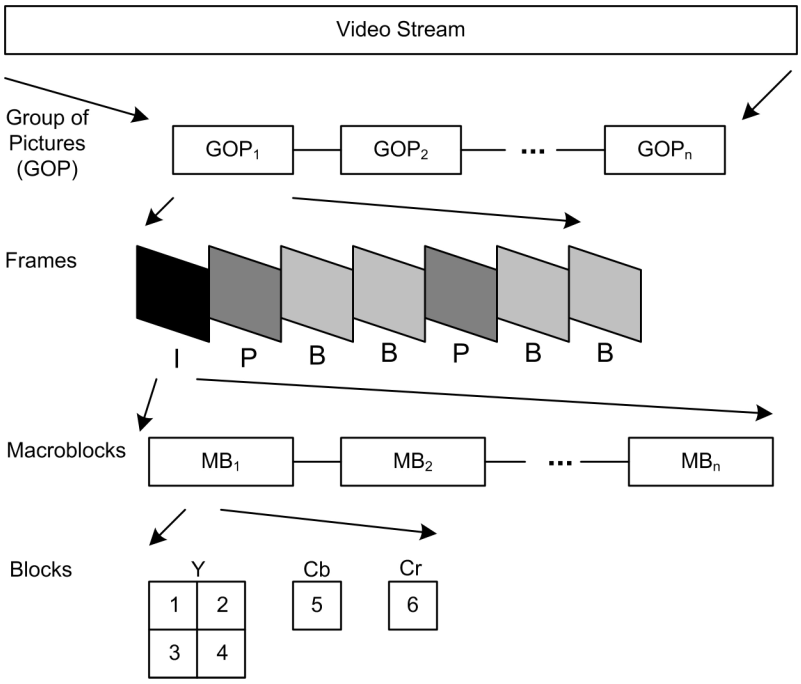


Figure 2-4. Structure of a Typical MPEG Video Stream

MPEG video uses three main techniques to achieve compression: *motion compensation*, *transform coding*, and *entropy coding*. *Motion compensation* is an inter-frame compression technique that works at a macroblock level and is used to eliminate temporal redundancy. In video sequences, it is highly likely that adjacent frames have a fairly large amount of common data and therefore searching for similar blocks of data in adjacent pictures and coding only the differences between these blocks and the vectors representing the relative position of these blocks can achieve frame data compression to a great extent. I-frames do not use motion compensation since they are "standalone" frames used as reference frames for the other two types of frames. P-frames use forward

prediction with I-frames as reference frames. B-frames may use forward as well as backward prediction with I- and P-frames as reference frames.

Transform coding is an intra-frame compression technique used in MPEG to exploit spatial redundancy in a video stream. It includes the application of Discrete Cosine Transform (DCT), quantization, and run-length encoding on each coded 8x8 block of pixels. DCT is a method of decomposing a block of data into a weighted sum of spatial frequencies and each of these spatial frequency patterns has a corresponding coefficient associated with it. The main goal of using DCT is to achieve decorrelation of data so that the coefficients can be coded independently. Quantization uses the frequency response of the human eye to an advantage and divides the DCT coefficients so that the resultant coefficients are in a limited range of allowed values. Quantization assists in compression by causing most of the AC coefficients from the DCT operation to result in zero, which helps in the next step of transform coding i.e. run-length encoding. Run-length encoding converts the block coefficients into a series of run-level pairs that indicate the number of zeroes preceding a non-zero coefficient. Finally, *entropy coding* or variable-length codes (VLCs) are used to further achieve compression.

In order to obtain the original frame data back, the above compression process is reversed at the decoder. A generic video decoding model is shown in Figure 2-5 and it consists of the following main steps [21]: Variable Length

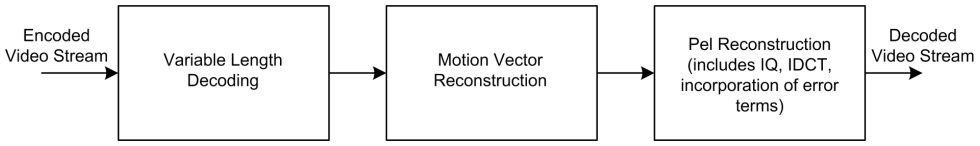


Figure 2-5. Generic Video Decoding Model

Decoding to decode the VLCs, reconstruction of motion vectors, and pixel reconstruction, which comprises of Inverse Quantization (IQ), Inverse Discrete Cosine Transform (IDCT), and finally incorporation of error terms in the blocks.

3. LITERATURE REVIEW

This chapter gives an overview of previous work done in DVS on video decoding. Some of the more important algorithms are discussed in detail, which also sets a platform for our proposed DVS technique that is presented in the next chapter.

3.1 Classification of DVS Schemes for Video Decoding

Previous works done with DVS on video decoding that fit in the scope of this thesis primarily differ in two ways: (1) Granularity at which the voltage/frequency setting is varied, and (2) Workload prediction mechanism. A video stream may be considered as consisting of various logical building blocks such as GOPs, frames, macroblocks, and blocks in decreasing order of size and a choice must be made regarding the granularity at which the voltage/frequency setting should be changed periodically. In making this decision, it is important to take into account the characteristics of a generic video stream e.g. a GOP consists of different types of frames that have different characteristics and purpose in the video stream. Also, the workload processing requirement of each of these different types of frames may be different depending on the characteristics of a particular video clip in question. An I-frame always consists of the maximum possible number of macroblocks, with all blocks in a macroblock coded, and

therefore would ordinarily lead to the maximum processing requirement. On the other hand, the processing requirement for a P- or B- frame would depend to a great extent on the characteristic of the particular video clip e.g. whether the video clip is high-motion, low-motion, has many scene changes etc. Thus, intuitively, it seems that using a GOP granularity to change the voltage/frequency setting may not be an appropriate choice since trying to meet the frame deadline for the most complex frame in terms of processing requirement may lead to slack times in case of frames that are relatively less complex, which ultimately implies lost opportunity in power reduction through DVS. Another point that must be taken into account when making this decision is the time required to make the transition from one voltage/frequency setting to another which may take between 70μs to 140μs [3, 23]. If a finer granularity such as a block or a macroblock is chosen for making the voltage/frequency transitions then not only does it cause an overhead but whether the benefit obtained in terms of power-savings when selecting such a fine granularity is significantly valuable or not should also be taken into consideration. Most of the prior works done [23, 24, 26-31] with DVS on video decoding have chosen a frame as a logical unit when making the voltage/frequency transitions with the exception of one scheme [25] that makes these transitions on a per-GOP basis. Currently to our knowledge, there are no DVS algorithms using a finer granularity than a frame for making voltage/frequency transitions.

The second important factor, using which DVS schemes for video decoding can be classified, is the workload prediction mechanism. The greater the accuracy of the prediction scheme, the higher is the possibility that an optimum power saving is obtained that is as close as possible to the ideal DVS case, where each prediction hits right on target with no over or under predictions. It has been found [24] that there is a strong correlation between frame sizes and the decoding times of a frame. Most of the schemes proposed earlier [23-27] use some form of frame size vs. decoding time relationship to predict the future workload. While some methods preprocess a clip to create fixed linear size vs. decoding time relationships, others dynamically change this relationship to adapt to a video clip as it is being played.

3.2 Prior Work – DVS Schemes on Video Decoding

Son et al. [25] in their work have proposed two per-GOP based DVS schemes. Since in their work, they have shown one of their methods to be more effective than the other, we only explain that method here. We refer to this method as *GOP-Decoding Time Prediction* (*G-DTP*) in our thesis. When using this method, the voltage/frequency for an upcoming GOP is decided upon by predicting its workload in terms of the decoding time that will be required. A size vs. decoding time relationship is used in order to do so. The size of the upcoming GOP in terms number of bytes is obtained from the sum of the frame sizes in the

GOP. A moving average of the Decoding Time per Byte (DTPB) required for each frame type is found out and updated at the end of every GOP. This is used along with the upcoming GOP size to predict the decoding time for the next interval. Since the voltage scaling interval used in this scheme is fairly coarse-grained i.e. a per-GOP interval, it is very likely that this scheme does not take full advantage of DVS by possibly setting the voltage/frequency levels in a higher range in order to meet video QoS requirements in terms of frame deadlines, as mentioned previously, thereby consuming more power than what might be consumed if a finer-grained voltage/frequency setting interval is used. This inference has been established as correct in the work presented in [26].

The works presented in [23, 27] use pre-established and fixed frame size vs. decoding time relationships derived from data collected for a particular video clip to predict the decoding time for an upcoming frame and thus use a frame-based granularity for setting the voltage/frequency. This technique will be referred to as *Frame-Fixed Equation* (*F-FE*) in this thesis. Another scheme presented in [23] also uses a frame-based granularity for setting the voltage/frequency level but the prediction mechanism used is different. In this second scheme, a frame is divided into two halves (possibly either depending on the number of macroblocks in the frame or the frame size in bytes). The voltage/frequency level for the first half of the frame is decided based on a pre-established frame size vs. decoding time relationship mentioned above,

distinguished according to the frame type, and the length of the preceding frame of the same type. A complexity ratio between the upper and lower halves of the previous frame is also determined. The length of the second half of the frame is estimated using the product of the complexity ratio of the previous frame and the length of the first half of the current frame. The voltage/frequency setting for this estimated length is then decided upon based on the pre-established frame size vs. decoding time relationship. The processor speed is adjusted if the estimate exceeds or falls short of the actual frame time.

A DVS scheme referred to as *Frame-Dynamic Equation* (*F-DE*) in this thesis, implemented in [26], again uses a frame size vs. decoding time relationship to vary the voltage/frequency setting at a frame-based granularity. Although this scheme also predicts the frame decoding time using the incoming frame size, it does not use a fixed relationship as in the previously mentioned schemes. On the contrary, it uses a dynamically changing equation that varies and adjusts itself depending on the history of previously recorded frame size vs. decoding time values. This type of an approach certainly has an advantage over the schemes using fixed relationships since such a scheme can more easily adapt itself to videos with different characteristics and particularly to videos with high-motion or frequent scene changes.

With the basic essence of the previous schemes proposed for DVS on video decoding summarized, it can be seen that all of the above schemes have one

common feature: they all require the upcoming frame size as an input to their prediction algorithms e.g. the GOP-based schemes [25] require the sum of the sizes of all frames in the subsequent GOP, and while [23], [27], and [26] differ in the way they predict the upcoming frame decoding time, all of these schemes also require the frame size or length as an input to their prediction algorithms. Also, while some methods [25, 26] establish a size vs. decoding time relationship dynamically, other schemes [23, 27] need to establish this relationship *before* the actual decoding of the stream begins. Such an approach requiring *a priori* knowledge of the stream may not be suitable to real-time video applications.

With this background, there could be one more way of categorizing DVS on video decoding schemes into *off-line* and *on-line* schemes. DVS techniques could be termed as on-line if they do not require preprocessing to input information to the DVS algorithm. On the other hand, they are classified as off-line if they do require preprocessing.

There are also some other techniques that have been proposed for DVS on video decoding and these are explained next. The work in [28] has proposed an off-line scheduling algorithm for preprocessing stored MPEG video. The idea is for the media servers to preprocess the video streams before delivering them and transmit the streams to the clients along with the voltage scheduling to be used at the client. This method is impractical since it requires knowledge of client hardware. Besides, it may not be able to support real-time video streams. Choi et

al. [29] have proposed a scheme in which they divide the frame decoding period into a frame-dependent and a frame-independent part and scale voltage accordingly. However, as mentioned in their work, it is possible for a single inaccurate prediction to propagate across frames and degrade video quality. The work in [30] has presented an integrated algorithm that uses architectural adaptation along with workload prediction for DVS to reduce power consumption during video decoding, while [31] has proposed the use of buffers with DVS in multimedia applications. These methods cannot be compared on the same level as the methods in focus in this thesis but rather can be looked upon as potential improvements for the previously explained methods.

Our motivation for this thesis mainly stems from trying to overcome the limitation found in currently proposed DVS methods for video decoding, namely the *a priori* knowledge of video stream parameters, and to propose a technique that would not have such a requirement.

4. Frame-Data Computation Aware DVS Scheme

We propose a DVS scheme called the *Frame-data Computation Aware (FDCA)* method that is an on-line DVS method and uses a frame-based granularity for changing the voltage/frequency settings.

The principal idea is to extract useful frame-related information *while* decoding a frame to estimate the decoding time for the frame, which is then used as input to the DVS algorithm. This way, there is no need to carry out any preprocessing to obtain the required information.

4.1 Restructured MPEG decoder

It can be recalled from Figure 2-5 in Chapter 2 that the main steps in the video decoding process include [21] Variable Length Decoding (VLD), reconstruction of motion vectors (MC), and pixel reconstruction, which comprises of inverse quantization (IQ), inverse discrete cosine transform (IDCT), and incorporation of error terms in the blocks (Recon). Ordinarily, an MPEG decoder carries out decoding on a per-macroblock basis and the above mentioned steps are repeated for each macroblock until all macroblocks in a frame are exhausted as shown in Figure 4-1.

With this decoding cycle for a frame in the video stream, it is not possible to obtain any valuable information that can be used for predicting the decoding

time for a frame since such information is not available before the actual decoding of a frame begins.

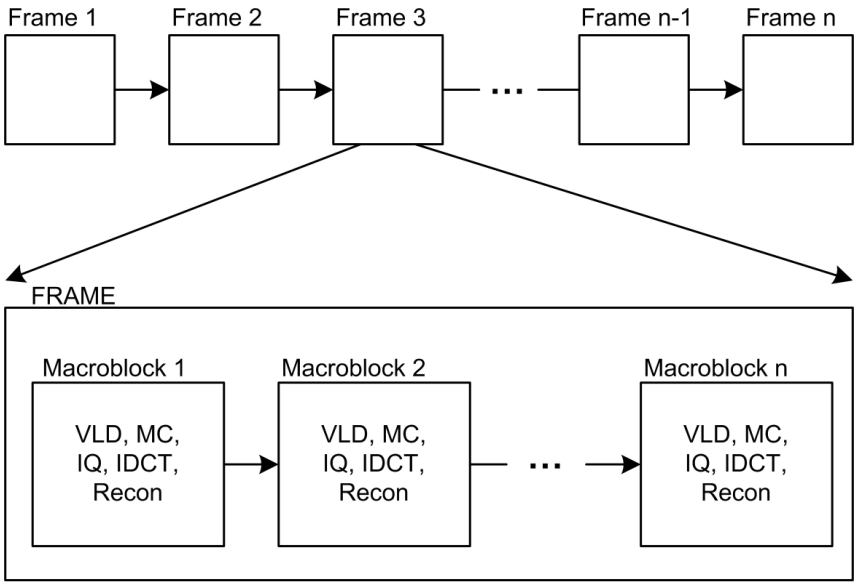


Figure 4-1. Decoding Cycle in a Generic MPEG Decoder

On studying the MPEG video standard, and the decoder at source code level, it was found that the Variable Length Decoding (VLD) step in the decoding process of a frame is potentially able to provide some required useful information e.g. `macroblock_motion_forward/macroblock_motion_backward` in the `macroblock_type` field of a macroblock header indicate whether the macroblock has a forward/backward motion vector defined or not, decoding the DCT coefficients indicates the number of coefficients that will require inverse quantization etc. Also, it is possible to break up the VLD step for the entire frame from the rest of the steps in the decoding process. Decoding can thus be regarded

as a process with two steps: Step (1) VLD for entire frame and, Step (2) All decoding operations in a frame following VLD.

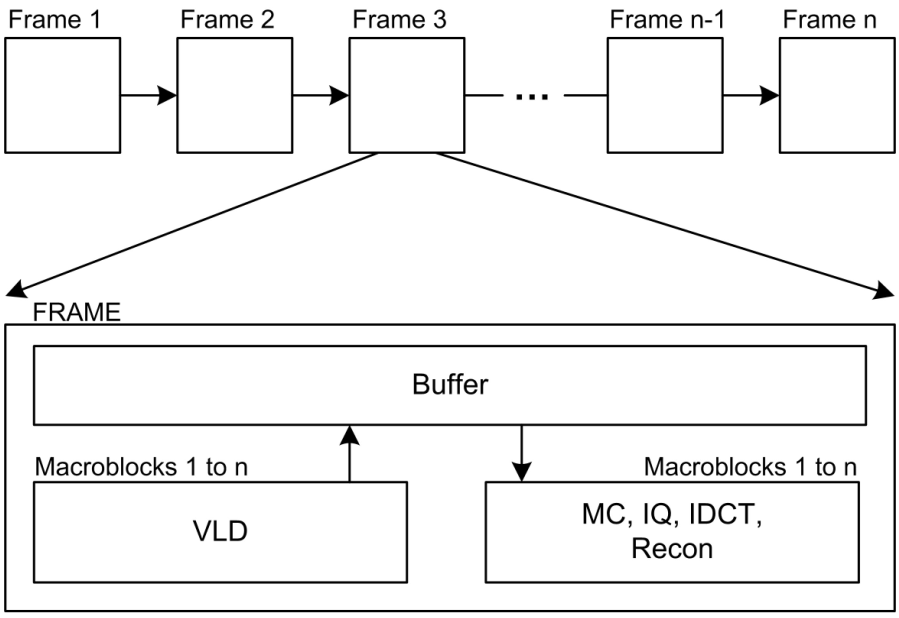


Figure 4-2. Decoding Cycle in Restructured Decoder for FDCA

We thus restructured the video decoder so that for a given frame, the VLD step for all macroblocks in the frame is carried out ahead of the rest of the decoding steps as shown in Figure 4-2. The decoding cycle of Figure 4-1 demands only the information from the current macroblock being decoded and some selected information from the previous macroblock to be stored at any given time. However, with the decoding cycle of Figure 4-2, it now becomes necessary to store this information for all macroblocks in a given frame. The original Berkeley *mpeg_play* MPEG-1 decoder [35] used in our experiments contains data

structures to store information related to the various logical building blocks of a video stream. To adapt to the decoding cycle of Figure 4-2, we modified these data structures. Since the number of macroblocks can differ for different frames, we modified the data structure pertaining to a frame to now contain a linked list of macroblocks in that frame, while the modified data structure for a macroblock points to an array of blocks in the macroblock in question. Aside from this information, the VLD step in the restructured decoder also stores additional information that can be used for this DVS approach for the frame being decoded. This additional information is in the form of: (1) total number of motion vectors to be calculated (nbrMV) in a frame, (2) total number of block coefficients in a frame (nbrCoeff), (3) total number of blocks on which to carry out IDCT (nbrIDCT), and (4) the number of blocks to perform error term correction on (nbrRecon). These parameters for the entire frame are summed up and stored in the data structures as the VLD step for the frame progresses. An example of the original and modified data structures is shown in Appendix A.

While the first step of the restructured decoder i.e. VLD stores all the necessary decoded information in these data structures, the various operations performed in step 2 of decoding read back this information to reconstruct the frame.

4.2 Workload Estimation

One of the most important requirements for applying DVS is the prediction of future workload. In order to achieve this, we further divide step 2 of the restructured decoder into "unit operations". These logical unit operations can be distinguished as: (1) calculation of motion displacements for one motion vector, (2) performing inverse quantization on one coefficient in a block, (3) performing inverse discrete cosine transform on one block, and (4) incorporating error terms for one block. These unit operations were decided upon after studying the source code of the decoder and identifying sections of code that will be executed for different operations. An example of a unit operation in the form of a section of code that calculates the motion displacements for one motion vector is shown in Appendix A. While the concept of identifying such sections will largely be similar for different decoders, the actual sections will vary depending on the implementation of a particular decoder.

In the decoder that we used in our simulations, the inverse discrete cosine transform executes one function when more than one coefficient in a block is non-zero and another optimized function when only one coefficient in a block is non-zero. Thus, even though logically, inverse discrete cosine transform is one "unit operation", we identified two different sections of code that would be executed depending on the number of coefficients in a block. A similar situation exists for the unit operation of incorporating error terms. Different sections of code are

executed depending on whether a block is intra-coded or non-intra coded and also depending on the number of motion vectors present. However, for the sake of keeping the explanations clear, we group such sections of code under one logical "unit operation".

Every unit operation executes the same block of code each time that operation is performed and therefore will require similar number of cycles. Therefore, a moving average to smooth out variations can be maintained, at frame level, of the cycles required for each of the unit operations after the VLD step. These parameters consist of the number of cycles required for (1) reconstructing one motion vector (AvgTimeMC), (2) carrying out IQ on one coefficient (AvgTimeIQ), (3) performing IDCT on one block on pixels (AvgTimeIDCT), and (4) incorporating error terms on one block of pixels (AvgTimeRecon).

Using data collected in the VLD step i.e. nbrMV, nbrCoeff, nbrIDCT, and nbrRecon, and the parameters mentioned in the paragraph above, it is now possible to estimate the number of cycles that will be required for frame decoding after the VLD step by simply multiplying the corresponding parameters and taking the sum of the products. We also maintain a moving average of the prediction error (PredError), which is the difference between the actual number of cycles required for step 2 of frame decoding and the estimated cycles obtained from the sum of products above, and use this as an adjustment to the final estimated decoding time for a frame. The window size for all moving averages in

our algorithm was considered to be 5. The cycles for the unit operations are not grouped according to the frame type because, as previously stated, the same block of code will be executed regardless of the type of the frame. The prediction error however, is grouped depending on the frame type.

In order to apply DVS, the time required for performing the VLD step is noted and using the frame rate for the video stream, the time available for step 2 of the frame is calculated. The estimated number of cycles for a frame is then used to apply DVS by selecting the bwest voltage/frequency setting that would meet the frame deadline. The VLD step is performed at the highest voltage/frequency setting available to leave as much time as possible to perform DVS during the more computationally intensive tasks after VLD. Thus, video decoding using the *FDCA* scheme would typically look as shown in Figure 4-3 below. Figure 4-4 gives an algorithmic description of the *FDCA* scheme.

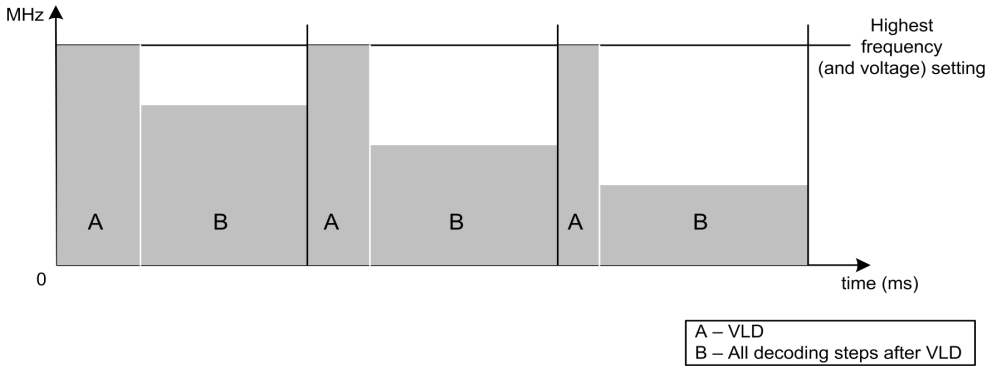


Figure 4-3 A Typical Example of a Video Stream Decoded with FDCA

For each frame I of frame type f_type:

1. Perform VLD at the highest voltage/frequency setting available and obtain nbrMV, nbrCoeff, nbrIDCT, and nbrRecon.

2. At the end of VLD, determine the time available (tRemaining) and estimate the time (in cycles) required for all decoding steps after VLD using data gathered in step 1 above and step 5 below.

$$EstDecodeTime_i = (nbrMV_i \times AvgTimeMC) + (nbrCoeff_i \times AvgTimeIQ) + (nbrIDCT_i \times AvgTimeIDCT) + (nbrRecon_i \times AvgTimeRecon) + (PredError_{f\_type})$$

3. Determine the lowest frequency setting f that would satisfy $EstDecodeTime_i$

```
f = f_lowest;
while (f = f_highest) {
   if  ((EstDecodeTime_i / f) < tRemaining) break;
  else f = next higher setting of processor speed; }
```

4. Accordingly, set the corresponding voltage/frequency setting

```
SetFreq(f) {MHz = f; Vdd = FreqToVolt(f);}
```

5. Update AvgTimeMC, AvgTimeIQ, AvgTimeIDCT, AvgTimeRecon, and $PredError_{f\_type}$ (window size = n)

$$AvgTimeMC_i = ActualTimeMC_i / nbrMV_i;$$
$$AvgTimeMC_{i+1} = ?_{\text{last n frames}}\ AvgTimeMC / n;$$

$$AvgTimeIQ_i = ActualTimeIQ_i / nbrCoeff_i;$$
$$AvgTimeIQ_{i+1} = ?_{\text{last n frames}}\ AvgTimeIQ / n;$$

$$AvgTimeIDCT_i = ActualTimeIDCT_i / nbrIDCT_i;$$
$$AvgTimeIDCT_{i+1} = ?_{\text{last n frames}}\ AvgTimeIDCT / n;$$

$$AvgTimeRecon_i = ActualTimeRecon_i / nbrRecon_i;$$
$$AvgTimeRecon_{i+1} = ?_{\text{last n frames}}\ AvgTimeRecon / n;$$

$$ActualTime_i = AcutalTimeMC_i + ActualTimeIQ_i + ActualTimeIDCT_i + ActualReconTime_i;$$

$$PredError_{f\_type} = EstDecodeTime_i - ActualTime_i;$$
$$PredError_{i+1} = ?_{\text{last n frames}}\ PredError / n;$$

Figure 4-4. Algorithm for FDCA DVS Approach

Our approach is similar to the one proposed in [32] in that, we carry out VLD for the entire frame ahead of the other decoding steps. However, there are some key differences between the two methods: (1) Their work takes the worst case execution time of frames into consideration and tries to *lower* the overestimation as much as possible by using various frame parameters. Thus, their method not only causes an overhead due to decoder restructuring, but also causes an overestimation of decoding time. Our method on the other hand, takes a "best effort" estimation approach by using moving averages in the estimation. (2) Our ultimate goal is to use the decoding time estimation for applying DVS. Therefore, unlike their method, we choose to not buffer the entire frame (which may possibly lead to some delay and in turn more number of cycles and therefore more power consumption) to find out the frame size for estimation of decoding time for the VLD step. Instead, we start with VLD right away, thus also bypassing the preprocessing step that is required in their method.

5. SIMULATION ENVIRONMENT

The simulation environment used in the evaluation of our DVS scheme is explained in this chapter. It consists of a system that includes the *SimpleScalar* [33] simulator and the *Wattch* [34] tool along with the *mpeg_play* MPEG-1 video decoder [35]. The integrated simulation environment is shown in Figure 5-1 [26].

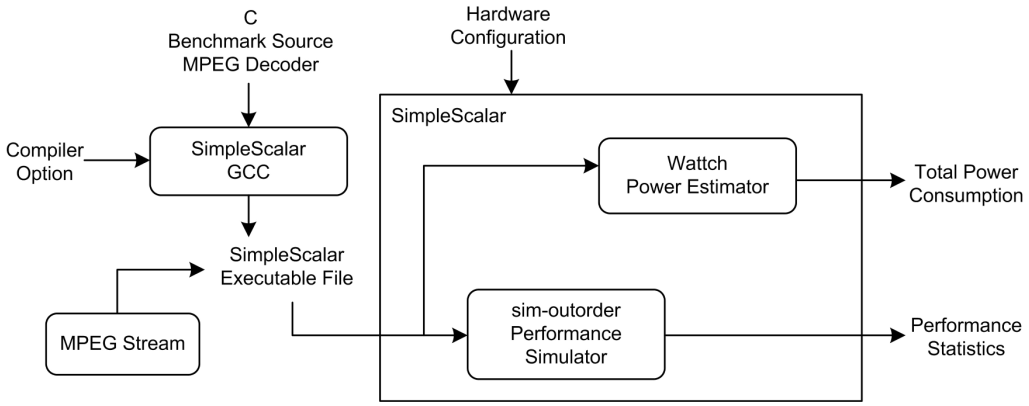


Figure 5-1. Integrated Simulation Environment for Evaluating DVS Schemes

SimpleScalar is an architectural simulator that provides a detailed simulation environment for various types of modern microprocessors. The simulator includes a proxy system call handler facility that can be used to make operating system-like calls (termed *syscall*s) from the application program. *Wattch* is an architectural-level power analyzer, which can be interfaced with the *SimpleScalar* simulator to estimate CPU power consumption. The Berkeley

mpeg_play decoder was run on this simulation framework for a selected set of sample workload streams.

5.1 Simulated DVS Algorithms

In our opinion, a fair evaluation of the performance of our proposed DVS *FDCA* scheme can be presented by not only providing the results from our DVS method, but also by comparing the results from our technique with those of the DVS schemes proposed in prior related work. We therefore selected three different methods [26], that we felt were of significance with respect to the core of this thesis and which could be compared on the same level as our scheme. The other methods selected and implemented along with our *FDCA* scheme are: (1) *GOP-Decoding Time Prediction* (*G-DTP*) from [25], (2) *Frame-Fixed Equation* (*F-FE*) from [23], and (3) *Frame-Dynamic Equation* (F-DE) from [26]. Apart from these four methods, we also simulated a *No DVS* and an *Ideal* scenario. In the *No DVS* scenario, the simulations were carried out at the highest voltage/frequency setting available assuming that the processor is not a variable voltage processor. The results presented are relative to the *No DVS* case. On the other hand, in the *Ideal* case, we assume an ideal workload prediction mechanism.

5.2 Workload Video Streams

Three video streams were selected as a set of fairly diverse sample workload that could represent different types of video stream characteristics in general. Table 5-1 gives a summary of the workload streams used in our simulations. Clip 1 is referred to as *Children* in this thesis and is a low-motion clip about a public message regarding childcare. Clip 2, called *Red's Nightmare*, is an animation video while Clip 3 is a very small segment from the motion picture *Under Siege* and represents a high-motion video.

Table 5-1. Workload Videos Sample Set Used in DVS Simulations

Characteristics	**Children**	**Red's Nightmare**	**Under Siege**
Generic Movement	Low	Medium	High
Resolution	320 x 240 pixels	320 x 240 pixels	352 x 240 pixels
Frame Rate (fps)	29.97	25.0	30.0
Total Frames	899	1211	731
I Frames	62	41	123
P Frames	238	81	122
B Frames	599	1089	486
Pred. Eqn. for *Frame-Fixed* (Decoding Time in cycles – tCycles)	tCycles = 88.8 x Frame size + 10^6	tCycles = 53.9 x Frame size + (2 x 10^6)	tCycles = 69.6 x Frame size + (2 x 10^6)

The decoding time characteristics, essentially representing the processing requirement of frames in each of these clips in terms of the number of cycles required, are shown in Figure 5-2 through Figure 5-4. A high-motion video would

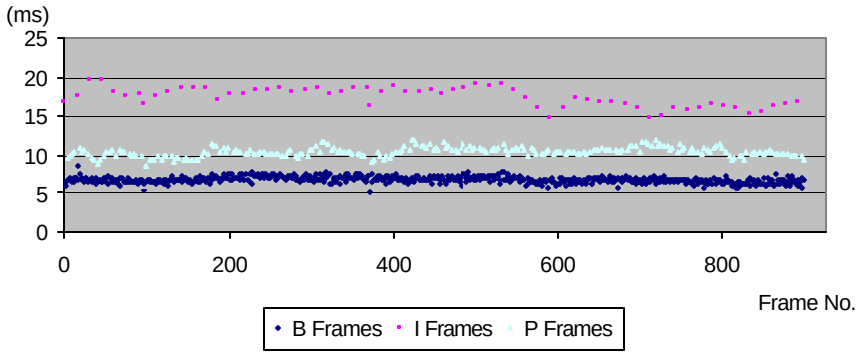


Figure 5-2. Decoding Time Characteristics for *Children* Clip

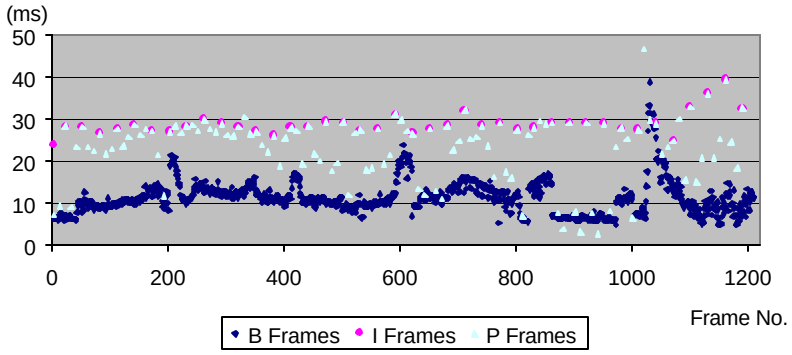


Figure 5-3. Decoding Time Characteristics for *Red's Nightmare* Clip

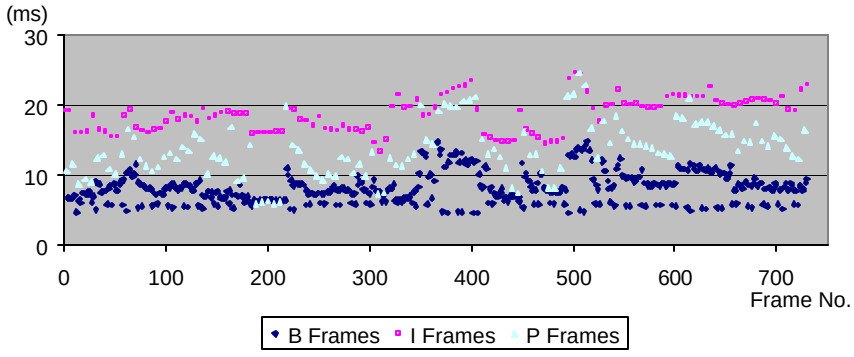


Figure 5-4. Decoding Time Characteristics for *Under Siege* Clip

mostly have an erratically varying processing requirement even within the same frame type and this can be clearly seen from the peaks in the graphs for the high-motion video of *Under Siege*. On the other hand, for a low-motion video like *Children*, the number of cycles required for decoding do not fluctuate much and a distinct demarcation for the three different types of frames can be seen from the graph. This data in Figure 5-2 through Figure 5-4 was also used to simulate the *Ideal* DVS case.

5.3 Simulator Framework Adaptations

We configured the *SimpleScalar/Wattch* framework to resemble a five-stage pipelined processor as is likely to be used in most portable devices. Also, to be as close as possible to a practical situation, we used 13 voltage/frequency settings to simulate a Strong-ARM-like processor [23]. The voltage/frequency settings used in our simulations are listed in Table 5-2. *SimpleScalar* was also further modified to include the code for setting the appropriate voltage/frequency level and the workload prediction algorithms. The original unmodified *mpeg_play* decoder was used for simulation of *GOP-Decoding Time Prediction* (*G-DTP*), *Frame-Fixed Equation* (*F-FE*), and *Frame-Dynamic Equation* (*F-DE*) methods, while the restructured decoder was used for simulation of our *FDCA* method. The decoders were modified to include the DVS system calls to the simulator. A DVS

Table 5-2. Voltage/Frequency Settings Used in DVS Simulations

Voltage (V)	**Frequency (MHz)**
0.79	59
0.861667	75
0.933334	91
1.005001	107
1.076668	123
1.148335	139
1.220002	155
1.291669	171
1.363336	187
1.435003	203
1.50667	219
1.578337	235
1.65	251

system call will modify the voltage/frequency setting currently used by the processor if required. These system calls were also used to find out the number of cycles required for decoding a frame and updating other data used in an algorithm during the decoding process. Two system calls were made in *G-DTP*, *F-FE*, and *F-DE*: one at the start of a frame and the other at the end of the frame. In the *FDCA* method there are also other system calls made to update data related to cycles for unit operations. It was assumed that the overhead involved in making the voltage/frequency transitions is trivial since the time required for making

these transitions is negligible as compared to the interval granularity i.e. GOP and frame granularity used for making these transitions.

5.4 Performance Evaluation Parameters

We used three parameters to evaluate the performance of our *FDCA* scheme and also to compare the results with those from the other DVS methods mentioned earlier. The first and obvious parameter used was power consumption during the decoding process. The power consumption is indicated in terms of the *average power consumption per frame* relative to the *No DVS* case. The performance of the methods is evaluated in terms of the *error*, which is defined as the ratio of the standard deviation of inter-frame playout times [36] to playout interval. This parameter basically defines how well a DVS method was able to meet frame deadlines, as also how smooth a video clip played with the given method. The third parameter used is the *deadline misses*. More number of missed deadlines will obviously cause a degradation of the video quality. But an important point to consider here is also the *degree of deadline misses* that indicates the extent by which the deadlines were missed and therefore this factor is also evaluated in our simulation results.

6. SIMULATION RESULTS

This chapter presents the results from the simulations carried out and an analysis of the same on the four DVS schemes: *GOP-Decoding Time Prediction* (*G-DTP*), *Frame-Fixed Equation* (*F-FE*), *Frame-Dynamic Equation* (*F-DE*), and our proposed method, *FDCA* scheme using the performance parameters explained in the previous chapter.

6.1 Relative Average Power Consumption per Frame

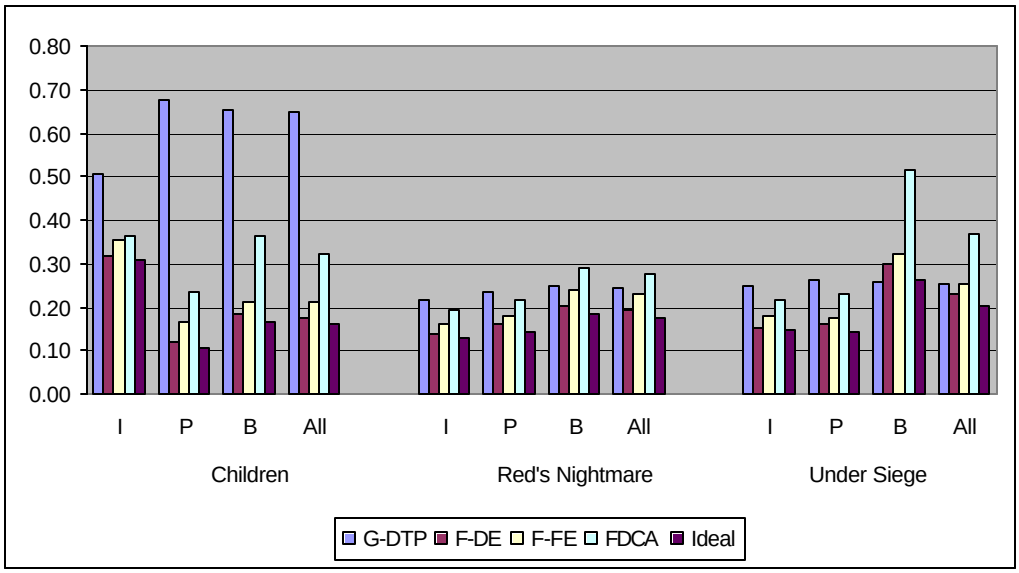


Figure 6-1. Relative Average Power Consumption per Frame

The graph of the relative average power consumption per frame, relative to using no DVS is shown in Figure 6-1. The graph shows that the *Ideal* case, which is used as a reference, as expected, consumes the least amount of power since it uses an ideal workload prediction mechanism and gives an average of

about 82% power savings for the workload streams and processor voltage/frequency settings used in our simulations. The *Ideal* case indicates the maximum power savings that can be achieved for a given processor setting in the simulations. If the decoding time characteristics for the *Children* clip in Figure 5-2 are observed, then it can be seen that the average time required for I-frames is much larger than that required for P- and B-frames in general. When a video clip with this type of decoding characteristics is used with the *GOP-Decoding Time Prediction* method, then based on a discussion in an earlier chapter, it would be easy to understand why this method consumes much more power for the *Children* clip as compared to all the other methods. In setting the same voltage/frequency level for different types of frames in a single GOP, this method possibly creates slack periods for frames with lesser processing requirement and ends up consuming more power giving only about 35% power savings for the *Children* clip and an average of about 62% power savings over the three different clips. Simulations for all the methods for different clips except the case above give a performance that is quite comparable to the ideal case. Another observation is that the *FDCA* method consumes more power than the other three methods in general. This is because the restructured decoder used in the *FDCA* scheme requires more number of cycles than the original unmodified decoder used in the other schemes. Our simulations on the sample streams show that *FDCA* has an average of about 12% overhead compared to the original decoder in terms of the number of cycles

required. Therefore, this overhead represents lost opportunity to save power using DVS. In addition, *FDCA* stores frame-related data in the VLD step and loads the data back again during the rest of the steps, which leads to about 9-14% higher data cache miss rate as compared to the original decoder. However, the *FDCA* method, which is an on-line method, is still able to provide an average of about 68% of power saving which is quite substantial.

Among the *off-line* methods, *Frame-Dynamic Equation* performs the best giving about an average of 80% power saving, while *Frame-Fixed Equation* gives an average of about 77% power saving for the three workload streams used.

6.2 Accuracy in Terms of Error

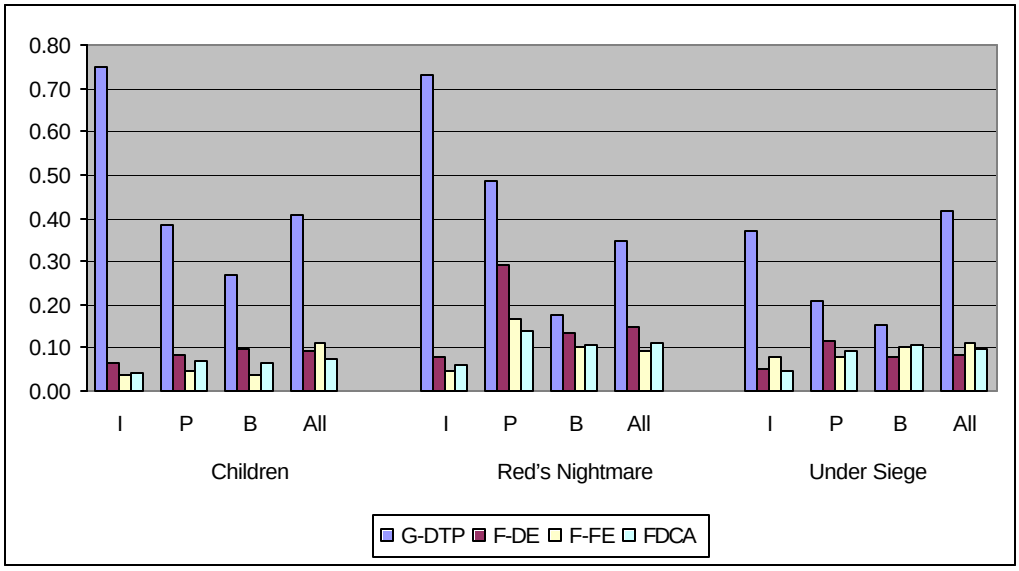


Figure 6-2. Error for various DVS approaches

Figure 6-2 shows the error for the various DVS approaches simulated. This parameter is a measure for the accuracy of the prediction algorithm as also of the perceptual quality of the video. Once again, *GOP-Decoding Time Prediction* does not perform as well as the other methods and gives an overall average error of about 38.9%. The *FDCA* scheme has the best performance in this category with an average error of about 9.4%, which possibly is because of the fact that it uses detailed frame-specific parameters to estimate the decoding time of a particular frame. *Frame-Dynamic Equation* and *Frame-Fixed Equation* also give a good result with an average error of about 10.8% and 10.5% respectively. Although these two methods give comparable performance, it should be noted that the *Frame-Fixed Equation* method would have an edge over the *Frame-Dynamic Equation* method since *Frame-Fixed Equation* not only uses the known frame sizes but also uses a pre-established frame size vs. decoding time relationship that the *Frame-Dynamic Equation* builds as it decodes the stream.

6.3 Deadline Misses

The next parameter based on which the simulated DVS approaches may be compared is the percentage deadline misses for the three clips and these results are shown in Figure 6-3. In this category, again because of the extent of *a priori* knowledge that the *Frame-Fixed Equation* method uses, it performs the best giving the smallest percentage of average deadline misses of 5.6%. *Frame-*

Dynamic Equation and *FDCA* give a comparable result of 16.04% and 13.4% respectively, with *FDCA* thus performing better than the *Frame-Dynamic Equation* method. Even though P-frames for the Children clip for *FDCA* cause about 35% deadline misses, it is found that the deadlines are missed by only an

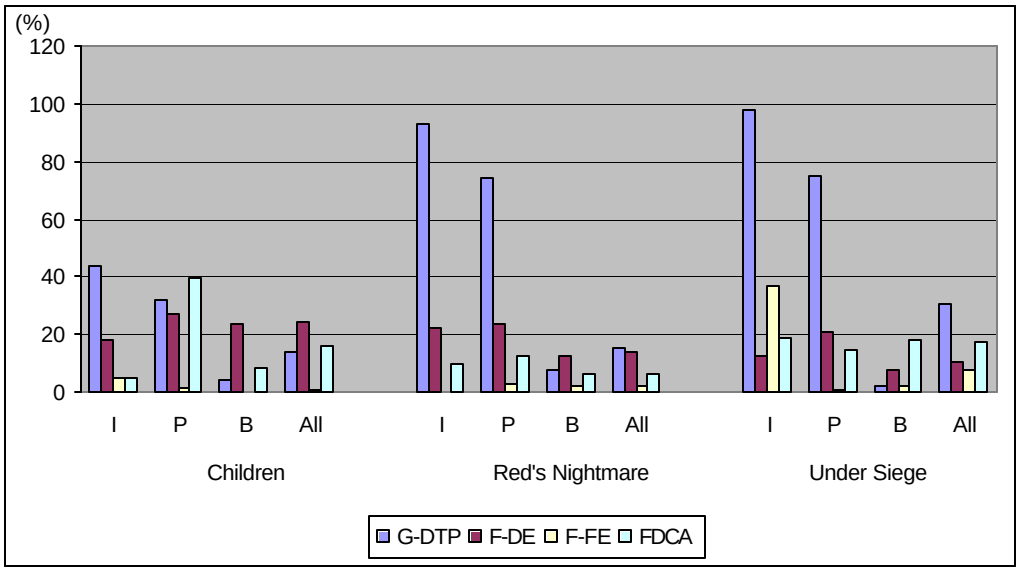


Figure 6-3. Percentage of Deadline Misses for Various DVS Approaches

average of about 5% which is negligible. *GOP-Decoding Time Prediction* causes the most number of deadline misses with an overall percentage of about 39.26%.

It would also be interesting to find out not only the percentage of deadline misses as above but also the extent by which these deadlines were missed. These results are shown in Figure 6-4. It can be seen from the figure that the *GOP-Decoding Time Prediction* not only has the most number of deadline misses but it

also causes most of these deadlines to be missed by more than 40%. It would not be surprising therefore that the use of this DVS method would give a comparatively poor perceptual video quality. *Frame-Dynamic Equation* and *FDCA* have missed deadlines but these deadlines are within 10-20% of the

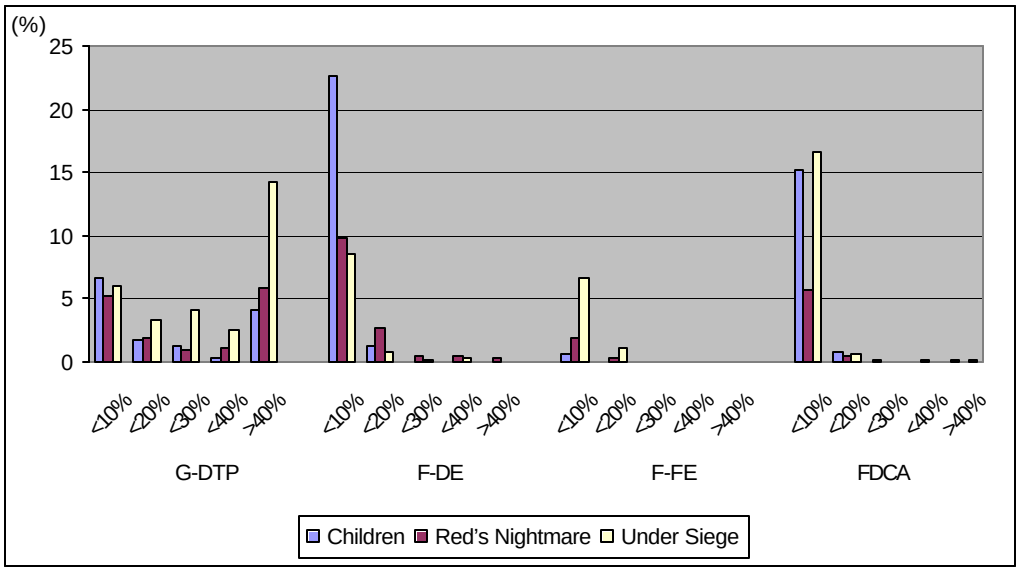


Figure 6-4. Degree of Deadline Misses for Various DVS Approaches

specified target deadline and therefore are not likely to harm the perceptual quality. *Frame-Fixed Equation* also has most of the deadline misses within the 10% range.

7. CONCLUSION

This thesis proposed a DVS scheme called the *Frame-data Computation Aware (FDCA)* scheme that is capable of predicting frame sizes in a video stream using frame-specific parameters extracted from the stream while decoding it. This scheme overcomes the difficulty faced by other DVS schemes proposed earlier of having to preprocess video streams to obtain information needed for their algorithms, thus allowing it to be classified as an on-line DVS scheme that is effective for and adaptable to real-time video scenarios. Our scheme was compared against some previously proposed schemes considered important by us such as the *GOP-Decoding Time Prediction*, *Frame-Direct Equation*, and *Frame-Dynamic Equation* schemes. We found that although the power saving obtained from the *FDCA* scheme was not as high as that provided by the *Frame-Direct Equation* and *Frame-Dynamic Equation* schemes, but the savings were still comparable, and our scheme was able to provide an average power saving of about 68%, which is quite substantial. The *FDCA* scheme also fared very well in terms of QoS parameters and provided an average error of about 9.4%, which was the best among the methods simulated here. The percentage of missed deadlines was an average of 13.4% and the extent by which these deadlines were missed was within 20% of the target deadline. This indicates that this method will be able to provide a good perceptual quality even with videos with different

characteristics. Thus, this approach is very suitable for portable multimedia devices, which require low-power consumption.

There are a number of areas in which future work could be done to further the research done in this thesis. Firstly, new ways could be explored to find more accurate prediction mechanisms for the unit operations in video decoding, in particular for IDCT, as also the effect of using a finer granularity voltage setting interval e.g. a macroblock or a block level interval. It would be interesting to actually implement the algorithm in a real-world hardware setting and evaluate the results for the same. Secondly, the effect of using the *FDCA* method, which already gives a good performance, with other DVS methods proposed such as architectural adaptations [30] would be an interesting area to investigate. Lastly, the use of these DVS methods which currently only target to lower the CPU power consumption, with other DVS methods to target other areas of a system such as memory, network interface etc. as also the effect of network jitters on these algorithms could be studied as future work.

BIBLIOGRAPHY

[1] J. Lorch, "A Complete Picture of the Energy Consumption of a Portable Computer", *Masters Thesis, Computer Science, University of California at Berkeley*, Dec. 1995.

[2] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-Power CMOS Digital Design", *IEEE Journal of Solid-State Circuits 27*, 4 (1992), 119-123.

[3] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen, "A Dynamic Voltage Scaled Microprocessor System", *IEEEJ. Solid-State Circuits*, vol. 35, no. 11, pp. 15714580, November 2000.

[4] T. Ishihara, and H. Yasuura, "Voltage Scheduling Problem for Dynamically Variable Voltage Processors", *In Proceedings of International Symposium on Low Power Electronics and Design (ISLPED '98)*, 1998, pp. 197402.

[5] J. Pouwelse, K. Langendoen, and H. Sips, "Dynamic Voltage Scaling on a Low-Power Microprocessor", *7th ACM International Conference on Mobile Computing and Networking (Mobicom)*, pp. 251-259, Rome, Italy, July 2001.

[6] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for Reduced CPU Energy", *In Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1994, pp. 13 23.

[7] T. Pering, T. D. Burd, and R. B. Brodersen, "The Simulation and Evaluation for Dynamic Voltage Scaling Algorithms", *In Proceedings of International Symposium on Low Power Electronics and Design (ISLPED '98)*, August 1998, pp 76 81.

[8] D. Grunwald, P. Levis, K. I. Frakas, C. B. Morrey III, and M. Neufeld, "Policies for Dynamic Clock Scheduling", *In Proceedings of the 4th Symposium on Operating System Design and Implementation*, October 2000.

[9] E. Chan, K. Govil, and H. Wasserman, "Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU", *In Proceedings of the First International Conference on Mobile Computing and Networking (MOBICOM '95)*, pages 13-25, November 1995.

[10] K. Flautner, S. Reinhardt, and T. Mudge, "Automatic Performance-Setting for Dynamic Voltage Scaling", *In Proceedings of the 7th Conference on Mobile Computing and Networking (MOBICOM '01)*, 2001.

[11] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli, "Dynamic Voltage Scaling and Power Management for Portable Systems", *In Proceedings of Design Automation Conference*. pp.524-529, 2001.

[12] T. Pering, and R. Brodersen, "Energy Efficient Voltage Scheduling for Real-Time Operating Systems", *In Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium RTAS '98*, Work in Progress Session (Denver, CO, June 1998).

[13] P. Pillai, and K. G. Shin, "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems", *In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, (New York October 21-24 2001), G. Ganger, Ed., vol. 35, 5 of ACM SIGOPS Operating Systems Review, ACM Press, pp. 89-102.

[14] J. Lorch, and A. J. Smith, "Improving Dynamic Voltage Scaling Algorithms with PACE", *In Proceedings of the International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS)*, June 2001.

[15] J. S. Seng, D. M. Tullsen, and G. Z. Sai, "Power-Sensitive Multithreaded Architecture", *International Conference on Computer Design*, pp. 199-206, September 2000.

[16] D. Shin, J. Kim, and S. Lee, "Low-Energy Intra-Task Voltage Scheduling Using Static Timing Analysis", *In Proceedings of Design Automation Conference*, pages 438-443, June 2001.

[17] J. Pouwelse, K. Langendoen, and H. Sips, "Energy Priority Scheduling for Variable Voltage Processors", *In Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2001.

[18] J. Flinn, and M. Satyanarayanan, "Energy-aware Adaptation for Mobile Applications", *In Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP)*, pages 48-63, December 1999.

[19] C.-H. Hsu, U. Kremer, and M. Hsiao, "Compiler-Directed Dynamic Voltage/Frequency Scheduling for Energy Reduction in Microprocessors", *In*

Proceedings of the International Symposium on Low-Power Electronics and Design, August 2001.

[20] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau, "Architectural and Compiler Strategies for Dynamic Power Management in the COPPER Project", *International Workshop on Innovative Architecture*, January 2001.

[21] K. Patel, B. Smith, and L. Rowe, "Performance of a Software MPEG Video Decoder", *In Proceedings of the First ACM International Conference on Multimedia*, pages 75-82, August 1993.

[22] J. L. Mitchell, D. L. Gall, and C. Fogg, "MPEG Video Compression Standard", *Chapman & Hall*, 1996.

[23] J. Pouwelse, K. Langendoen, R. Lagendijk, and H. Sips, "Power-Aware Video Decoding", *Picture Coding Symposium (PCS '01)*, Seoul, Korea, April 2001.

[24] A. Bavier, B. Montz, and L. Peterson, "Predicting MPEG Execution Times", *SIGMETRICS / PERFORMANCE '98, International Conference on Measurement and Modeling of Computer Systems*, pp. 131-140, June 1998.

[25] D. Son, C. Yu, and H. Kim, "Dynamic Voltage Scaling on MPEG Decoding", *International Conference of Parallel and Distributed System (ICPADS)*, June 2001.

[26] E. Nurvitadhi, B. Lee, C. Yu, and M. Kim, "A Comparative Study of Dynamic Voltage Scaling Techniques for Low-Power Video Decoding", *International Conference on Embedded Systems and Applications*, June 23-26 2003.

[27] J. Shin, "Real Time Content Based Dynamic Voltage Scaling", *Masters Thesis*, Information and Communication University, Korea, January 2002.

[28] M. Mesarina, and Y. Turner, "Reduced Energy Decoding of MPEG Streams", *ACM/SPIE Multimedia Computing and Networking*, 2002.

[29] K. Choi, K. Dantu, W. –C. Chen, and M. Pedram, "Frame-Based Dynamic Voltage and Frequency Scaling for a MPEG Decoder", *In Proceedings of*

International Conference on Computer Aided Design, pp. 732-737, November 2002.

[30] C. J. Hughes, J. Srinivasan, and S. V. Adve, "Saving Energy with Architectural and Frequency Adaptations for Multimedia Applications", *In Proceedings of the 34th International Symposium on Microarchitecture*, 2001.

[31] C. Im, H. Kim, and S. Ha, "Dynamic Voltage Scheduling Technique for Low-Power Multimedia Applications Using Buffers", *In Proceedings of the 2001 International Symposium on Low Power Electronics and Design, ACM Press*, 2001, pp. 34-39

[32] P. Altenbernd, L. O. Burchard, and F. Stappert, "Worst-Case Execution Time Analysis of MPEG-2 Decoding", *12th Euromicro Conference on Real-Time Systems*, Stockholm, 2000.

[33] D. C. Burger, and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", *Technical Report CS-TR-97-1342*, University of Wisconsin, Madison, June 1997.

[34] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations", *In Proceedings of the 27th International Symposium on Computer Architecture (ISCA '00)*, June 2000, pp. 83-94.

[35] Berkeley MPEG Tools. http://bmrc.berkeley.edu/frame/research/mpeg

[36] Y. Wang, M. Claypool, and Z. Zuo, "An Empirical Study of RealVideo Performance Across the Internet", *In Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, November 2001.

APPENDICES

APPENDIX A. Code Samples from Restructured Decoder

A-1 Sample Data Structures from Original *mpeg_play* and Corresponding Restructured Decoder Data Structures

```
/* Macroblock structure in original decoder. */

typedef struct macroblock {
  int mb_address;                    /* Macroblock address.*/
  int past_mb_addr;                  /* Previous mblock address.*/
  int motion_h_forw_code;            /* Forw. horiz. motion vector code.*/
  unsigned int motion_h_forw_r;      /* Used in decoding vectors.*/
  int motion_v_forw_code;            /* Forw. vert. motion vector code.*/
  unsigned int motion_v_forw_r;      /* Used in decdoinge vectors.*/
  int motion_h_back_code;            /* Back horiz. motion vector code.*/
  unsigned int motion_h_back_r;      /* Used in decoding vectors.*/
  int motion_v_back_code;            /* Back vert. motion vector code.*/
  unsigned int motion_v_back_r;      /* Used in decoding vectors.*/
  unsigned int cbp;                  /* Coded block pattern.*/
  BOOLEAN mb_intra;                  /* Intracoded mblock flag.*/
  BOOLEAN bpict_past_forw;           /* Past B frame forw. vector flag.*/
  BOOLEAN bpict_past_back;           /* Past B frame back vector flag.*/
  int past_intra_addr;               /* Addr of last intracoded mblock.*/
  int recon_right_for_prev;          /* Past right forw. vector.*/
  int recon_down_for_prev;           /* Past down forw. vector.*/
  int recon_right_back_prev;         /* Past right back vector.*/
  int recon_down_back_prev;          /* Past down back vector.*/
} Macroblock;


/* Macroblock structure in restructured decoder. */

typedef struct Macroblock {
  int mb_address;                   /* Macroblock address.*/
  int motion_h_forw_code;           /* Forw. horiz. motion vector code.*/
            :
            :

  /*****Added for Restructured Decoder*/

  struct tpBlock *ptrArrBlocks;     /* Pointer to arr. of blocks in mb.*/
  short int dct_dc_y_past;          /* Past lum. dc dct coefficient.*/
  short int dct_dc_cr_past;         /* Past cr dc dct coefficient.*/
  short int dct_dc_cb_past;         /* Past cb dc dct coefficient.*/
  int recon_right_for;              /* Decoded horiz. forw. mv elem.*/
  int recon_down_for;               /* Decoded vertical forw. mv elem.*/
  int recon_right_back;             /* Decoded horiz. back mv elem.*/
  int recon_down_back;              /* Decoded vertical  back mv elem.*/
  BOOLEAN mb_motion_forw;           /* forward vector present.*/
  BOOLEAN mb_motion_back;           /* backward vector present.*/
  struct tpMacroblock* nextMacroblock; /* pointer to next mb in frame.*/
  BOOLEAN slice_start;               /* macroblock marks start of slice.*/

  /*****End Additions for Restructured Decoder*/
} Macroblock;
```

```
/* Picture structure in original decoder. */
typedef struct pict {
  unsigned int temp_ref;              /* Temporal reference.             */
  unsigned int code_type;             /* Frame type: P, B, I             */
  unsigned int vbv_delay;             /* Buffer delay.                   */
  BOOLEAN full_pel_forw_vector;       /* Forw. vectors specified in full
                                         pixel values flag.              */
  unsigned int forw_r_size;           /* Used for vector decoding.       */
  unsigned int forw_f;                /* Used for vector decoding.       */
  BOOLEAN full_pel_back_vector;       /* Back vectors specified in full
                                         pixel values flag.              */
  unsigned int back_r_size;           /* Used in decoding.               */
  unsigned int back_f;                /* Used in decoding.               */
  char *extra_info;                   /* Extra bit picture info.         */
  char *ext_data;                     /* Extension data.                 */
  char *user_data;                    /* User data.                      */
} Pict;


/* Picture structure for Restructured Decoder. */
typedef struct pict {
  unsigned int temp_ref;               /* Temporal reference.*/
  unsigned int code_type;              /* Frame type: P, B, I*/
            :
            :
  /*****Additions for Restructured Decoder*/

  struct tpMacroblock* headMacroblock; /*pointers for linked list*/
  struct tpMacroblock* prevMacroblock;
  struct tpMacroblock* curMacroblock;
  struct tpMacroblock* tailMacroblock;

  BOOLEAN slice_start;         /*indicates if start of slice encountered */
  /*Variables added to collect data*/
  int nbrMBCount;              /*for reconstruction of motion vectors*/
  int nbrMB_ZeroMV;
  int nbrMB_OneMV;
  int nbrMB_TwoMV;
  int coeffCount;              /*for inverse quantization*/
  int nbrICB_coeffmorethan1;   /*for inverse discrete cosine transform*/
  int nbrICB_coeffequalto1;
  int nbrIMB;                  /*for incorporation of error terms*/
  int nbrReconBiMB_zflag1;
  int nbrReconBiMB_zflag0;
  int nbrReconPMB_zflag1;
  int nbrReconPMB_zflag0;
  int nbrReconBMB_zflag1;
  int nbrReconBMB_zflag0;
  int nbrSkippedMB;

  /*****End Additions for Restructured Decoder*/
} Pict;
```

A-2. Motion Vector Calculation in Original Decoder.

This serves as an example of a “unit operation” for our FDCA DVS algorithm.

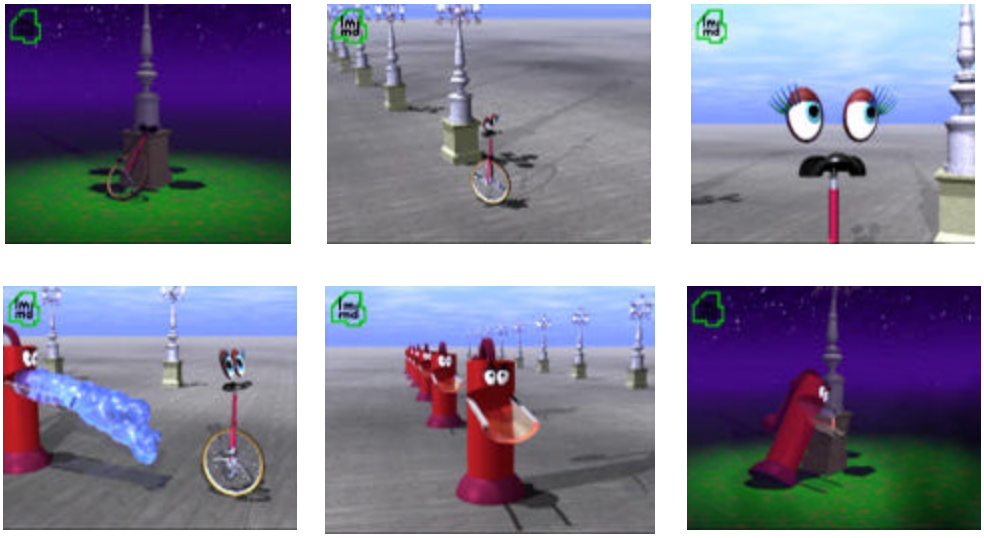
```
/*
 *----------------------------------------------------------------
 *
 * ComputeVector --
 *
 *     Computes motion vector given parameters previously parsed
 *      and reconstructed.
 *
 * Results:
 *      Reconstructed motion vector info is put into recon_*
parameters
 *      passed to this function. Also updated previous motion vector
 *      information.
 *
 * Side effects:
 *      None.
 *
 *----------------------------------------------------------------
 */

#define ComputeVector(recon_right_ptr, recon_down_ptr,
recon_right_prev, recon_down_prev, f, full_pel_vector, motion_h_code,
motion_v_code, motion_h_r, motion_v_r)                              \
                                                                    \
{                                                                   \
  int comp_h_r, comp_v_r;                                           \
  int right_little, right_big, down_little, down_big;               \
  int max, min, new_vector;                                         \
                                                                    \
  /*The following procedure for the reconstruction of motion vectors \
    is a direct and simple implementation of the instructions given \
    in the mpeg December 1991 standard draft.                       \
  */                                                                \
                                                                    \
  if (f == 1 || motion_h_code == 0)                                 \
    comp_h_r = 0;                                                   \
  else                                                              \
    comp_h_r = f - 1 - motion_h_r;                                  \
                                                                    \
  if (f == 1 || motion_v_code == 0)                                 \
    comp_v_r = 0;                                                   \
  else                                                              \
    comp_v_r = f - 1 - motion_v_r;                                  \
                                                                    \
  right_little = motion_h_code * f;                                 \
  if (right_little == 0)                                            \
    right_big = 0;                                                  \
  else {                                                            \
    if (right_little > 0) {                                         \
      right_little = right_little - comp_h_r;                       \
      right_big = right_little - 32 * f;                            \
    }                                                               \
    else {                                                          \
      right_little = right_little + comp_h_r;                       \
```

```
    right_big = right_little + 32 * f;                          \
  }                                                             \
}                                                               \
                                                                \
down_little = motion_v_code * f;                                \
if (down_little == 0)                                           \
  down_big = 0;                                                 \
else {                                                          \
  if (down_little > 0) {                                        \
    down_little = down_little - comp_v_r;                       \
    down_big = down_little - 32 * f;                            \
  }                                                             \
  else {                                                        \
    down_little = down_little + comp_v_r;                       \
    down_big = down_little + 32 * f;                            \
  }                                                             \
}                                                               \
                                                                \
max = 16 * f - 1;                                               \
min = -16 * f;                                                  \
                                                                \
new_vector = recon_right_prev + right_little;                   \
                                                                \
if (new_vector <= max && new_vector >= min)                     \
  *recon_right_ptr = recon_right_prev + right_little;           \
                   /* just new_vector */                        \
else                                                            \
  *recon_right_ptr = recon_right_prev + right_big;              \
recon_right_prev = *recon_right_ptr;                            \
if (full_pel_vector)                                            \
  *recon_right_ptr = *recon_right_ptr << 1;                     \
                                                                \
new_vector = recon_down_prev + down_little;                     \
if (new_vector <= max && new_vector >= min)                     \
  *recon_down_ptr = recon_down_prev + down_little;              \
                   /* just new_vector */                        \
else                                                            \
  *recon_down_ptr = recon_down_prev + down_big;                 \
recon_down_prev = *recon_down_ptr;                              \
if (full_pel_vector)                                            \
  *recon_down_ptr = *recon_down_ptr << 1;                       \
}
```

APPENDIX B. Snapshots of Video Streams Used in Simulations

B.1 *Children* Clip

B.2 *Red's Nightmare* Clip



B.3 *Under Siege* Clip