AN ABSTRACT OF THE THESIS OF

Varin Vahia for the degree of Master of Science in Electrical and Computer Engineering presented on April 1, 2003.

Title: Hybrid Adaptive Controller for Resource Allocation of Real-rate Multimedia Applications.

Abstract approved

Redacted for Privacy

Molly H. Shor

Multimedia applications such as video streaming and Voice over IP are becoming common today with the tremendous growth of the Internet. General purpose operating systems thus are required to support these applications. These multimedia applications have some timing constraints that need to be satisfied for good quality. For example, video streaming applications require that each video frame be decoded in time to be displayed every 33.3 milliseconds. In order to satisfy these timing requirements, general purpose operating systems need to have fine-grained scheduling. Current general purpose operating systems unfortunately are designed to maximize throughput to serve traditional data-oriented applications and have coarse-grained scheduling and timers. Time Sensitive Linux (TSL), designed by Goel, et al., solves this problem with fine-grained timers and schedulers. The scheduler for TSL is implemented at a very low level. The controller that implements the algorithm for resource allocation is implemented at a higher level. This controller can easily be modified to implement new control algorithms.

Successful implementation of resource allocation to satisfy timing constraints of multimedia applications requires two problems to be addressed. First, the resources required by the application to satisfy the timing constraints should not exceed the total available resources in the system.

Second, the controller must adapt to changing needs of the applications and allocate enough resources to satisfy the timing constraints of each application over time. The first problem has been addressed elsewhere using intelligent data dropping with TSL. We focus on the second problem in this thesis.

We design a proportion-period controller in this thesis for allocating CPU to multimedia video applications with timing constraints. The challenges for the controller design include the coarse granularity of the time-stamp markings of the video frames, the unpredictable decoding completion times of the frames, the large variations in the decoding times of the frames, and the limit of the control actuation to positive values. We set up the problem in a state space. We design a predictive estimating controller to allocate the proportion of the CPU to a thread when its long term error is small. When the decoding process is running behind by more than a certain threshold, we switch to a different controller to drive the error back to a small value. This controller is the solution to a dynamic optimization LQR tracking problem.

Hybrid Adaptive Controller for Resource Allocation of Real-rate Multimedia Applications

By

Varin Vahia

A THESIS

Submitted to

Oregon State University

In partial fulfillment of
the requirements for the
degree of

Master of Science

Presented April 1, 2003
Commencement June 2003

Master of Science thesis of Varin Vahia presented on April 1, 2003.

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for Privacy

Director of the School of Electrical Engineering and Computer Science

Redacted for Privacy

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.

Redacted for Privacy

Varin Vahia, Author

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

Figure Page

LIST OF FIGURES (Continued)

Figure Page

Chapter 1

Introduction

Today people are using more multimedia computer applications in everyday life than ever before. This abundance of multimedia applications is due to the tremendous growth of the Internet. In addition, Central Processing Unit (CPU) speeds have increased manifold in the last decade. Also, processors like Intel's Pentium 4 and MMX-based processors were designed to support multimedia applications. Because of this, it is now possible to process these multimedia flows, in addition to common desktop applications, on general-purpose desktop computer systems.

Nowadays, it is very common to watch streaming-video applications, videoconference and voice chat with colleagues, friends or relatives on common household Personal Computers (PCs). These applications no longer require dedicated hardware. We can run these applications, along with common household applications like word processing, Internet browsing, etc., without saturating the processor. These multimedia applications are soft real-time applications, meaning that they have real-world timing constraints that must be satisfied in order to get decent quality. These requirements, however, are not stringent since occasional deadline misses can be tolerated. An example of a soft real-time application is video streaming. Videos generally run at 30 frames/second, since at a slower video frame rate the human eye notices that the action is not continuously updated. This implies that we need to process the video frames in time to be displayed every 33.33 ms. If this is not satisfied, we have to drop frames, and consequently the quality of the video degrades. Similarly, Voice-over-IP (VoIP) systems require a one-way delay of at most 150-200 ms, where only a fraction of that time is available for processing at the application and operating-system level.

These applications require good support from operating systems, as well as adequate hardware resources. Two approaches that have been used to satisfy such timing constraints and end-to-end

delay requirements include dedicated hardware designed for a particular application and the reservation-based schemes in real-time operating systems. These approaches perform well, but they are limited in the scope where they can be applied. Dedicated hardware is an expensive solution if the particular application is not the primary one to be solved. Reservation-based schemes generally do not fully utilize the available resources if the resource requirements of the tasks vary significantly over time. To avoid underutilization, Abeni, et al., designed a feedback control scheme to adapt the reservations in a real-time operating system based on on-line measurements of a task's usage of the CPU resource [1].

Even if current hardware is more than ready to satisfy these requirements, current general-purpose operating systems are not. Currently, general-purpose operating systems do not provide good support to applications with timing constraints. This is because current general-purpose operating systems are designed for throughput-oriented applications. They use coarse scheduling periods and transfer data in large packets or disk blocks. This design choice decreases the frequency of interrupts, resulting in fewer invocations of interrupt-handling routines and context switches. This approach reduces interrupt overhead and thus increases throughput, since with less overhead there is a larger percentage of real work accomplished. Unfortunately, this is achieved at the expense of timing guarantees. Coarse scheduling periods result in large waiting times for applications that need to be scheduled. If these applications' timing deadlines fall within the scheduling periods, their deadlines will be missed. Also, if some timing constraints have a granularity that is finer than the scheduling-period granularities, it will be impossible to satisfy these timing constraints. These design decisions were made when the processor's speed was the limiting factor in system performance. By now, however, processing power has become abundant. We can now modify general-purpose operating systems so that they will support both throughput-oriented applications and applications with stringent timing constraints.

In order to adapt general-purpose operating systems to support applications with timing constraints, two problems must be addressed. First, the resources that are required to address end-to-end delay requirements must not exceed the total available resources of the system over a particular

period of time. Second, each task in the system must be smartly allocated the right amount of resources, at the right time, to guarantee the end-to-end delay requirements.

The first problem can be addressed in a number of ways. We can restrict which new tasks to accept in the system (admission control) or tasks can dynamically adapt their resource requirements (selective data dropping) if available resources are not sufficient. Admission control is useful if tasks cannot adapt their requirements while still functioning acceptably. If tasks have timing constraints, admission control may cause problems if missed data is important. In most multimedia applications, resolution and/or frame-rate can be reduced without sacrificing too much quality if available resources are not sufficient. Krasic and Walpole have implemented selective priority-based adaptation of video quality to address this problem [2, 3]. In this thesis, we have assumed that an intelligent data-dropping scheme is implemented to prevent total resource requirements from exceeding total available resources.

We focus on the second problem of "smart" resource allocation. We extend the control design work of Goel, et al., who designed a proportion-period scheduler for network and CPU resource allocation [4, 5]. Goel, et al., modified the Linux operating system to support applications with end-to-end delay requirements. Their version of Linux is called Time Sensitive Linux (TSL). TSL has improved support for both time-sensitive and throughput-oriented applications. Goel, et al.'s contributions included the design of firm reprogrammable timers for accurate timing, the design of various preemptible kernel schemes, dynamic tuning of the TCP send buffer length for low output latency, the design of feedback-based schedulers, and the design of software oscilloscope for visualization. They implemented all these features in TSL. We focus on improving the methodology used for the feedback controller design.

Supporting time-sensitive applications on a general-purpose operating system has several advantages over designing dedicated hardware. General-purpose operating systems are inexpensive. Some of them, like Linux, are open source, which means that they are freely available with source code available for both study and modification. Linux has a large user and developer

base. Support for this operating system is easily available. Modifying it is also easy due to its well-documented Application Programming Interface (API).

Supporting time-sensitive applications on a general-purpose operating system is challenging. For time-sensitive applications to work, we need to satisfy their temporal requirements. As discussed earlier, real-time operating systems traditionally use reservation-based schemes to meet temporal requirements. In such a reservation-based scheme, an application must specify its timing requirements in terms of CPU resources. In general-purpose operating systems, implementing a reservation-based scheme is not easy because there are a lot of other applications that are running simultaneously on the system. These other applications do not specify their CPU needs. We can not predict how much CPU they will use. Hence, we can not predict how much CPU will be available in advance. Thus we cannot guarantee any particular reservation of CPU to the time-sensitive applications. In addition, a video application's CPU resource requirements may vary over the length of the video. We ran some tests on MPEG video streams and found that the processing times of different frames can vary by a factor of four within a single video. Thus, a reservation-based scheme will waste considerable resources if we allocate for the worst-case requirements. In addition, a video's processing requirements may depend on the computer's hardware configuration and are not generally known *a priori*. To avoid underutilization, the CPU allocation to these applications needs to be dynamically adjusted based on their current CPU requirements. We can infer these requirements by measuring the progress made by these applications. Control system techniques can then be used very effectively to infer the processing time requirements of these time-sensitive applications based on measurements of the application's progress.

In the next section, we describe a streaming-video application in more detail as an example of a time-sensitive application. Then, in Section 1.2, we describe the low-latency requirements of these flows and how TSL satisfies these requirements. In Section 1.3, we describe feedback scheduling and control challenges in more detail. In Section 1.4, we describe the organization and contribution of this thesis.

1.1 A Streaming-Video Application

In this section, we describe a streaming-video application in more detail, as a particular case of a time-sensitive application. Figure 1.1 shows the architecture of a streaming-video application.

Video frames vary significantly in the processing resources required to decode them. The video source streams data at constant frame rate. The decoder, as its name suggests, decodes the encoded video frames. The display device displays the decoded video frames, at the same rate as at the source.

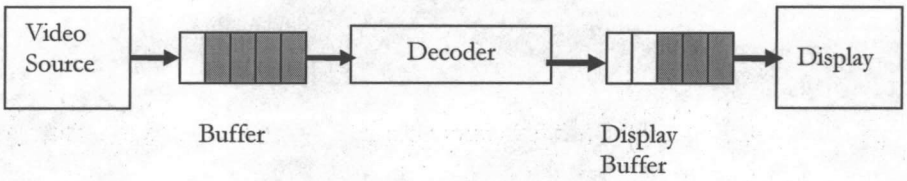


Figure 1.1: Video streaming application

The video source may be a video stored on a server or a live video, directly transmitted from its origin. If it were a stored video, we could determine *a priori* information about the processing resource requirements of the video for a particular computing platform. (This information is not generally measured and encoded for stored videos.) Examples of stored video on servers include news clips, movie and music-video clips available on various Internet sites. Examples of truly live videos include video conferencing, televised classes from various universities, etc. For live videos, we can not have any *a priori* information regarding the processing requirements of the video frames. We assume in this thesis that we do not have any prior information about the video that is being transmitted.

The video decoder requires the most CPU, and its CPU requirements vary from frame to frame. Our scheduler must allocate CPU to the decoder for frame decoding. Most video applications transmit data at around 30 frames/sec. If there were minimal, or no, buffering between the source, the decoder and the display, the decoder application would need to decode these MPEG frames at

precisely this rate and make them available for the display. Each frame would need to be processed in 33.3 ms. If suddenly the CPU needs of the video increased and we did not update the CPU allocation accordingly, the current frame would take more than 33.3 ms. The next frame would then arrive, in the case of real-time applications, and we would have to drop the current frame without buffering. The CPU requirements of individual frames can vary greatly depending on the content of the particular frame. If we have a measurement of the decoder's current progress, then, by comparing this measurement with its desired progress, we can assign the CPU accordingly. The correct allocation can be assigned using such a feedback control scheme. We have done testing on the MPEG video decoder. MPEG video streams have three types of frames, namely I, B and P frames. Our experiments show that the decoding CPU requirements vary from frame to frame. The I frames are the most CPU intensive. The P and B frames have decreasing CPU requirements, in that order. We will discuss this in more detail in Chapter 3, where we introduce the basics of the MPEG video standard.

In our implementation, the display device has its own process, which is given a constant CPU allocation. This process must run at constant time intervals. We need to make frames available to this display device in a buffer before each scheduling interval. If we fail to do so and the display buffer is empty, the display device will not have anything to display when it is run. This will result in a jittery video. Our scheduler needs to keep this display buffer filled with at least one frame.

1.2 Time Sensitive Linux (TSL)

In this section, we describe the low-latency requirements of time-sensitive flows and how Time Sensitive Linux (TSL) satisfies these requirements.

1.2.1 Low-Latency Applications

Video streaming, VoIP, and other time-sensitive applications are driven by real-world events. For example, one can design a video-streaming application to be driven by the constant-interval video frame arrivals, or by the constant-interval video-display requirements. In the first scenario, the kernel invokes the scheduler in response to an interrupt generated by the arrival of the video frame. The video frame is then delivered to an application such as a decoder. The application delivers its result, a decoded video frame, to the kernel. The kernel buffers the data until an external world response, such as a display occurs.

A *latent period* is a period of additional delay when there is no progress because a process is waiting for a resource to be allocated. Several latent periods are encountered during this transaction. The latent period from the arrival of the event to its delivery to the application is called the *input latency.* Similarly, the latent period from the application's completion to the delivery of the result to the external world is called the *output latency.* The kernel's design affects the input and output latencies. Time-sensitive applications require low input and output latencies. Time-sensitive applications are called *low-latency* applications. Kernel designs resulting in low input and output latencies reduce the end-to-end response time, ensuring proper handling of low-latency applications.

1.2.2 Latencies in Operating Systems

Various events that take place between the arrival of the data and the response of the application to the data are shown with their latencies in Figure 1.2.

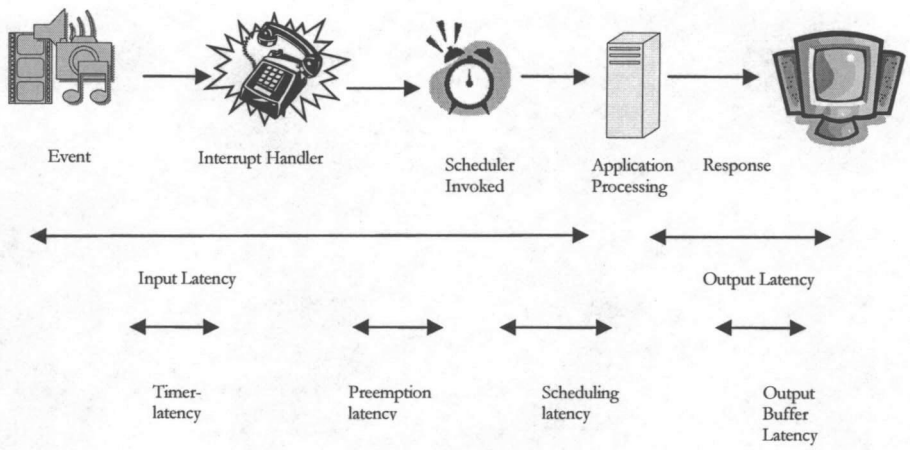


Figure 1.2 Low-latency applications' execution cycle

As shown in Figure 1.2, an application event, such as the arrival of a video frame, causes an interrupt in the kernel. The operating system handles this interrupt through an interrupt handler. The scheduler is then invoked to schedule the application in the CPU. The low-latency application starts running when it is scheduled to run. After the application has processed the event (e.g., decoded a video frame), it generates the result (e.g., a decoded frame ready to be displayed).

All of the above steps generate their own latencies in the response. As discussed earlier, input latency is the latency from when the external event occurs until the application receives the request through the kernel. Output latency is the latency from when a result is generated by the application until the delivery of the result to the external world. Further sub-classification of the input and output latencies is shown in Figure 1.2. Input latency is subdivided into timer latency, preemption latency and scheduling-policy latency. The buffer used to hide the rate mismatch between the application's data rate and the rate of the real-world device (e.g., video display) is the main cause of output latency. These latencies are described in short below. For a more detailed discussion on these latencies and their solutions, refer to [13].

Timer Latency

Timer latency is caused by the coarse granularity of the timer resolution. This is the largest source of latency in general-purpose operating systems. For example, Linux, by default, has a 10 ms. granularity in kernel and user-level applications. This means that every 10 ms the kernel timer interrupt will check for an application to execute. Hence, if an application's expiration event is not on a timer-tick boundary, the application may experience as much as 10 ms. of latency. TSL solves this problem by using firm timers. Firm timers are one-shot timers meaning they do not generate interrupts at predetermined fixed intervals. They are reprogrammed after each programmed period expires. We can choose the period according to an application's requirements. If an application has a period of 15 ms., we can program a one-shot timer to expire after this period. A soft timer with period of 5 ms. will generate three interrupts in this case, where only one is required.

Preemption Latency

Even if the interrupt is generated at the best time, it may not be serviced soon. This is because the kernel may be in a non-preemptible section. In a general-purpose operating system, when a thread enters the kernel, the thread becomes non-preemptible. This may cause a large latency if the thread is CPU intensive and takes a long time to run. TSL solves this problem by reducing the length of non-preemptible sections in the kernel.

Scheduling Latency

If the thread does not have the highest priority, it may not be scheduled immediately if some higher-priority process is present. This will cause scheduling latency. Real-time schedulers, such as proportion-period schedulers can be used to avoid this. But, real-time schedulers assume that all threads are full preemptible. Hence, such schedulers can be used when there are only short non-preemptible sections in a general-purpose operating system. TSL uses a proportion-period scheduler, together with short non-preemptible sections, to improve scheduling latency. We will look

at feedback scheduling in more detail in Section 1.3 since our optimal CPU allocator is part of this feedback scheduler.

TSL improves timer latency by 99.9%, preemption latency by 98% and TCP output-buffer latency by 95% compared to the standard Linux O/S [7].

1.3 Feedback CPU Scheduling

Traditionally, real-time scheduling algorithms, such as priority-based scheduling and proportional-period scheduling, have been used to satisfy the timing constraints of real-time flows. Proportional-period scheduling algorithms are traditionally not implemented in commodity operating systems since those generally have non-preemptive kernels. As discussed earlier, TSL overcomes these limitations by using firm timers and fine-grained preemptibility, together with a proportion-period scheduler.

In proportion-period scheduling, applications are assigned a certain proportion of the CPU over a certain period of time. Traditionally, this allocation is decided analytically by application programmers. This analytically-calculated allocation will be static since we can not modify the allocation while the application is running in this case. We will see in Chapter 3 that the processing requirements of multimedia flows vary greatly. Hence, it is difficult to find a good static allocation.

To solve this problem, Goel, et al., developed a feedback-based scheme to dynamically adjust the CPU allocation to an application based on measurements of its current progress. An application uses *time-stamps* to specify the timing requirements of its data units (frames) to the controller. These time-stamps correspond to the timing constraints of the application. For example, a video application that needs to process 30 frames/sec can timestamp each successive frame with a 33.3 ms. increment.

Our feedback controller compares these time-stamps with the real time and assigns an allocation of a certain proportion of the CPU for the next period so that the application keeps up with real time. The feedback controller adjusts the allocation dynamically. TSL and the proportion-period scheduler

provide the platform for this feedback controller and provide the fine temporal granularity that is required. The scheduler decides the order and timing of the processes within a time period.

Originally, TSL used a PI feedback controller to decide the CPU allocation. This implementation gave encouraging results. However, their PI feedback controller had limitations in how it was designed and the performance it produced. We will discuss the limitations in Chapter 2. We decided to design and test more advanced feedback control schemes to see if they would give more satisfactory results. In more recent work, Goel, et al., redesigned their controller [13].

In this thesis, we demonstrate an alternative methodology for the controller design for this problem. The details of the problem set up and general controller structure are discussed in Chapter 4. There are a number of challenges for the control design. The output that is measured is not continually updated but rather is updated only in response to a discrete event (the completion of a job). There is a random variation in a system parameter (the amount of resource required to complete a job). The value of the measured variable does not vary continuously, but rather by fairly large discrete (quantized) jumps. The CPU allocation is bounded between 0% and 100% (or less) of the CPU.

1.4 Contributions and Organization of Thesis

The rest of the thesis is organized as follows. Chapter 2 presents a brief overview of feedback control theory. It discusses both PI control and optimal control. Chapter 3 discusses the tests performed on MPEG videos. First, we give an overview of the MPEG video compression standard. Then we plot variations of decoding time of various MPEG video frames. We propose a MPEG decoding time predictor that can be constructed using this information. Chapter 4 provides an overview of the system and describes the modeling of this system. We set up our design problem as a dynamic optimization tracking problem in this chapter and compute an efficient numerical implementation. Chapter 5 discusses the implementation of the controller and results. We discuss

various approaches in this problem and state their advantages and disadvantages. We introduce a switching controller and various other implementations as we keep on refining our controller.

We set up and solve the control problem by switching between a predictive control solution, when there is a small error, and an optimal controller, when there is a larger error. We demonstrate several solutions that were developed on the way to that final solution, showing the advantages and shortcomings of each.

Chapter 2

FEEDBACK CONTROL THEORY

(PI AND OPTIMAL CONTROL)

The aim of this chapter is to present a brief overview of the feedback control schemes relevant to this thesis so that readers will have sufficient background to follow the later chapters. In Section 2.1, we discuss feedback control. In Sections 2.2 and 2.3, we discuss two particular approaches to designing a feedback controller – classical PI control and optimal control design. In Section 2.4, we discuss on-line estimation of system parameters and its use in the design of an adaptive controller. We apply these methods to our problem in later chapters. In Chapter 5, we will discuss the various controller implementations that we designed and tested to solve the particular problem described in this thesis.

2.1 Feedback Control

Feedback control schemes play a very important role in modern engineering because of their ability to detect the application's current requirements dynamically and change the input sequence accordingly. The *non-feedback (open-loop) system* shown in Figure 2.1 (a) has a pre-defined input. There is no mechanism to supervise the output. Because of this, we can not detect if the output deviates from the nominal desired behavior. This system does not have the ability to correct its behavior if it does not achieve the desired performance. On the other hand, the *feedback (closed-loop) system* shown in Fig 2.1 (b) is driven by two signals. One is the (external) input and the other is a feedback signal derived from the output. This feedback signal gives the feedback system the ability to self-correct in case its performance deviates from the desired performance.

The feedback system is driven by two signals. One is an externally-derived input signal, which is the desired system behavior. The other one is the feedback signal from a sensor. A "sensor" measures

the output signal, and the measured output is used as a feedback signal. The comparator compares this feedback with the input (i.e., the desired output) and generates an error signal, which is the difference between the desired output and the current output.

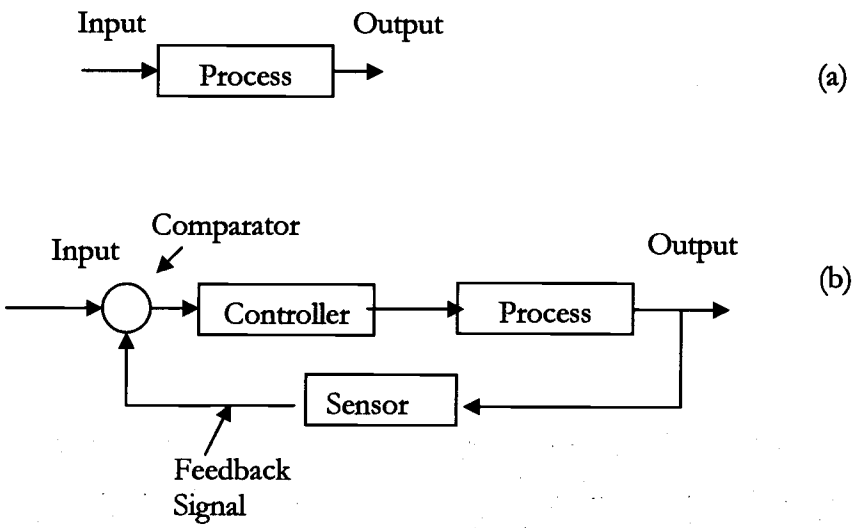


Figure 2.1 (a) Non-feedback system, (b) Feedback system

This error signal is the input to the controller. The controller decides how to change the input to the process in order to influence the output so that the error signal is reduced, so that the output attains the desired value. The controller can have many forms. For example, it may be basic PI controller or a more sophisticated optimal or adaptive controller. Our real-rate scheduler feedback loop is shown in Figure 2.2.

As shown in Figure 2.2, our real-rate controller consists of a comparator and an optimal allocator. The comparator, as its name suggests, compares the time-stamp of the last decoded video frame with the current wall clock time to see if the desired progress has been achieved. Depending on the error of this comparison, the controller (allocator) must decide what allocation to assign to the video decoder so that the decoding rate will match the desired rate.

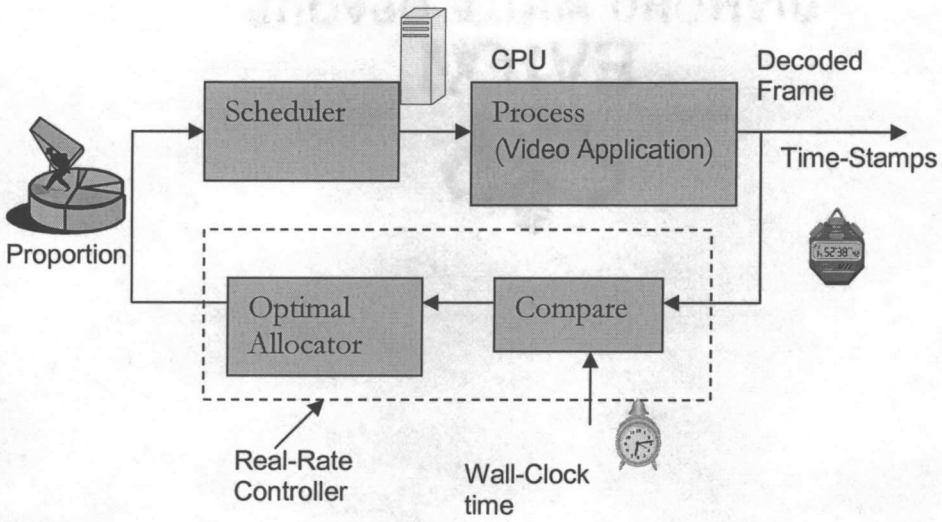


Figure 2.2 Feedback loop for real-rate controller

2.2 PI Control

The feedback control field includes many advanced techniques that can be used in various situations. One of the simplest commonly-implemented feedback control algorithms is the Proportional-Integral (PI) feedback controller. A PI controller computes the new input value to be applied to the system as a linear combination of the error at the current instant and the integration of the errors until the current instant. The designer must choose how to weight each type of error.

$$y_i = k_p e_{i-1} + k_i \sum_{n=1}^{i-1} e_n \qquad (2.1)$$

where,

i = current time
y = output
e = error
k_p = proportional gain
k_i = integral gain

As can be seen from Equation (2.1), we apply two gains, to the current error and the integration of the error, in a PI controller. The original scheduler in TSL used this PI controller to decide the

allocation for time-sensitive flows. We call these flows real-rate flows and the scheduler a real-rate scheduler. We will describe the PI controller used in TSL in later chapters.

2.2.1 Advantages and Limitations of PI Feedback Control

PI control is a good simple feedback scheme. By using two parameters for proportional and integral errors, the *type* of the system is increased by one. This means that the steady-state error to a particular type of input (step, ramp, etc.) reduces from constant to zero or from infinite to constant. PI control is quite simple to implement for SISO (Single-Input-Single-Output) systems. PI control is a good design choice if simplicity of implementation is desired.

Notice from Equation (2.1) that we need to choose the two parameters, the proportional gain and the integral gain, in order to design a PI controller. The most important challenge in designing a PI controller is to choose the proper values of the k_p and k_i parameters. There are some methods available to select these parameters, but simple trial-and-error can also be used for that purpose. The problem encountered by, Goel, et al., while designing their original PI controller was that a controller parameterization that was tuned to work well for step changes in the system input did not work well for impulse changes and vice versa.

One problem with PI control, and other feedback control schemes, is that the PI controller may suggest control input of any magnitude. However, in the real world, we have a limit on the magnitude of the inputs that we can apply. In the CPU, the processor has a certain speed limit beyond which it can not function. Hence, we need a bound on the applied input values so that they can not grow beyond the limit if the error becomes very large. This is standard problem with most real world systems. This problem has been studied in detail by the control system community and the addition of anti-windup mechanism is needed in the PI controller if the applied input value is not equal to the suggested input value. The input value of our system is also limited on the other end. One can not assign a negative CPU allocation if the video decoding gets ahead, which it may. The PI controller may generate a negative suggested input value. Again, we must bound the applied input values to be non-negative.

PI control works well for deterministic constant-parameter SISO systems with a single performance objective. If we have more than one performance objective, multiple inputs or outputs, significant randomness, or varying parameters in our system, PI control may not be useful. More advanced feedback design techniques are available to address these problems. Suppose we have multiple objectives or constraints in the problem. For example, suppose we would like to reduce the error and, in addition, limit the magnitude of the input. Then we can use an optimal control problem to derive a good controller. Suppose we have significant random noise in the system that affects the output signal. Then we can design a robust controller to suppress the effect of the noise on the output. Such robust controller can have PI structure but requires a model based design approach. Suppose we have an uncertain system or system with slowly changing behavior and do not know the exact model of the system. Then we can design an adaptive controller, which, as its name suggests, adapts itself to the changing system model.

We encountered several challenges while designing a controller for CPU allocation for the real-rate processes. First, there is a quantization in the error between the measured time-stamps and clock time. The measured output (the time-stamps) value has discrete jumps. The measured output is not continually updated. Second, a system parameter is random by nature.

Control system design techniques have not been used frequently until now for the design of computer systems. Our goal is to demonstrate that these schemes can be successfully applied to computer systems and that they have the potential to solve computer-system design problems. For our application of control, the quantization in measured output appears like noise, a random variation in a system parameter appears like model variation/uncertainty, and we have multiple performance objectives.

In the next section, we describe optimal feedback control.

2.3 Optimal Feedback Control

Optimal control design techniques provide a tool to design controllers that guarantee optimal, not just acceptable, system performance. In this approach, we select a performance index for the system to represent what is important to us and then design the controller to minimize this performance index. Selection of this performance index is the key design decision in an optimal control problem. This performance index is generally a function of the variables that we would like to minimize, such as an error or a function of an error, the control energy or a function of the control energy, the duration of the control, etc.

The optimal control problem is solvable if the system is completely controllable. This means that we can apply optimal control if we can transfer the system from one arbitrary state to another by applying some suitable input. If this is possible, then we can analytically solve the optimal control problem in many cases. If an analytical solution is not available, a numerical solution can be used for a controllable system. Controllability of a linear dynamical system can be tested if we know the state equation for a model of the system. Consider the linear constant parameter (time-invariant) state equation show below.

$$x_{k+1} = Ax_k + Bu_k \qquad (2.2)$$

Where x_k is the *(n x 1)* state vector and u_k is the *(m x 1)* control vector. *A (n x n)* and *B (n x m)* are the system matrices. This linear system is completely controllable if the rank of the matrix *Q* given below is *n*.

$$Q = \left[B \vdots AB \vdots A^2B \vdots \ldots\ldots \vdots A^{n-1}B\right]$$

We will show in Chapter 4 that our system is indeed completely controllable. This means we can successfully apply optimal control to our application.

Section 2.3.1 describes the optimal regulator problem in which the state vector is driven to zero, in its most general form. Section 2.3.2 specializes the problem to that of linear constant-parameter system with a quadratic cost functional. Section 2.3.3 extends this case to a tracking problem, where the system state is made to track some desired trajectory.

2.3.1 Formulation of Optimal Control Problem

In general, the system dynamics of a causal discrete-time control system can be described as shown as in Equation 2.3. A causal system is one where future values of input do not affect the present value of output.

$$x_{k+1} = f^k(x_k, u_k) \tag{2.3}$$

where x is an n-dimensional state vector, u is an m-dimensional control (input) vector and f is a function that determines the state at time $k + 1$ from the state and input at time k.

A general (scalar) performance index can be described as shown in Equation 2.4.

$$J_i = \phi(N, x_N) + \sum_{k=i}^{N-1} L^k(x_k, u_k) \tag{2.4}$$

Where *[i, N]* is the time interval over which we are interested in the behavior of the system, $\phi(N, x_N)$ is a positive function of the final time N and the state at the final time, and $L^k(x_k, u_k)$ is a positive function of the state and control input for each time instant k. The optimal control problem is to find the control input function u_k^* for the system that minimizes the performance index. This performance index is also called the *cost* or *penalty* of the system.

The problem stated here is the general form of the optimal control problem. Optimal control theory has developed closed-form solutions for certain classes of systems that are encountered very frequently in the real world and are amenable to closed-form solution. The form of the system and the performance index that are interesting to us and have a closed-form solution are a linear system

with a quadratic cost criterion. Our system is a linear system. We will describe the linear quadratic optimal regulator problem next.

2.3.2 Optimal Linear Quadratic Regulator (LQR)

As mentioned earlier, we have a linear system, in our case. For linear systems, a good choice of cost functional is a quadratic cost functional. The solution to the optimal control problem with a linear system equation and a quadratic cost is an easy-to-implement linear control law. Hence, even if the quadratic cost is not the best choice, it is most frequently used in order to avoid complex control laws. We present the LQR optimal control equations below.

The time-invariant discrete-time linear system is described by following state equation.

$$x_{k+1} = Ax_k + Bu_k$$

Where x_k is the state vector and u_k is the control vector. The performance index is given by

$$J_i = \frac{1}{2} x_N^T S_N x_N + \frac{1}{2} \sum_{k=0}^{N-1} (x_k^T Q_k x_k + u_k^T R_k u_k) \qquad (2.5)$$

Where all the matrices are symmetric and $S_N \geq 0$, $Q_k \geq 0$, $R_k > 0$. Here the symbol "$\geq$" means positive semi-definite and ">" means positive definite.

The optimal feedback control for this system, minimizing this performance index, is given by the following set of equations. This cost criterion penalizes non-zero values of the input and state vectors.

$$\begin{aligned} u_k &= -K_k x_k \\ S_k &= A^T [S_{k+1} - S_{k+1} B (B^T S_{k+1} B + R)^{-1} B^T S_{k+1}] A + Q \\ K_k &= -(B^T S_{k+1} B + R)^{-1} B^T S_{k+1} A \end{aligned} \qquad (2.6)$$

In this solution, we have assumed that A, B, *S, Q* and *R* are constant matrices and that the linear system is completely controllable.

Equation 2.6 for S_k is called the *discrete matrix Riccati equation* and must be solved backwards starting with the value S_N. Notice from these set of equations that the resulting feedback gain is time-varying. This time-varying feedback solution is not convenient to implement since it requires that we store the entire sequence of gain matrices K_k. Also, in our case, we do not know the final time when we will stop applying control. Calculating the gain in this case is impossible. One interesting property of the Riccati equation is that whenever *A, B, Q and R* are time-invariant and $N \to \infty$, the solution S_k converges to a steady-state solution *S*. This type of optimal control problem is called an *infinite-horizon* optimal control problem. The solution to this problem is given below.

$$
\begin{aligned}
u_k &= -Kx_k \\
S &= A^T[S - SB(B^T SB + R)^{-1} B^T S]A + Q \\
K &= -(B^T SB + R)^{-1} B^T SA
\end{aligned}
\qquad (2.7)
$$

The equation for *S* is called the *algebraic steady-state Riccati equation (ARE)*. The matrix *K* is the as a gain of the optimal controller. This controller can be represented as shown in Figure 2.3.

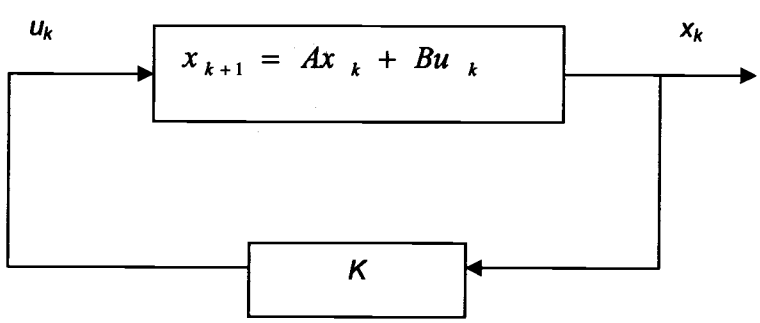


Figure 2.3 LQR feedback loop

Figure 2.3 shows the relative ease with which the LQR problem solution can be implemented, compared to the time-varying problem's solution. The current state x_k is fed back to the input through the linear feedback gain K, which is a constant. If the system state can be measured, or easily deduced from measurements, then this is easy to implement. Only one (matrix) gain need be stored. Because of the ease with which the LQR controller can be applied, it is popular with the control system community. This is one of the reasons why we chose this approach.

2.3.3 Infinite Horizon Linear Quadratic Tracking Problem

The cost functional for the optimal regulator problem is set up to minimize the system states or errors and the control action. A variant of this problem arises when we want the system states to follow some desired trajectory, minimizing the error between the state variable and the trajectory value. This type of control problem is called a tracking problem. The discrete-time tracking problem is explained in this section.

We will modify the cost functional for the tracking problem for the same linear system as before.

$$x_{k+1} = Ax_k + Bu_k$$

We want the state in our system to go to a non-zero constant value, rather than to zero. This can be accomplished by substituting the tracking error in the cost functional in place of the state variable. If we want the states to track the constant values described by the vector $\bar{x}$, then the cost functional can be modified as shown below.

$$J = \lim_{N\to\infty}\left\{\sum_{k=0}^{N}((x_k - \bar{x})^T Q_i (x_k - \bar{x}) + u_k^T R_k u_k)\right\}$$

Here, $\bar{x}$ is the desired tracking trajectory.

The solution to this problem is given by the following set of equations.

$$u(x_k) = Kx_k + K^v v$$

$$\begin{aligned} &\text{Where, } K = -(B^T SB + R)^{-1} B^T SA \\ &S = A^T [S - SB(B^T SB + R)^{-1} B^T S]A + Q \\ &K^v = (B^T SB + R)^{-1} B^T \\ &v = (A - BK)^T v + Q\bar{x} \end{aligned} \qquad (2.8)$$

By comparing Equation (2.8) with Equation (2.7), we can see that the LQ tracking controller has one term more than the LQR controller. This extra constant term corresponds to the constant tracking property required by the problem. The second term in the expression for *u* is called the feed-forward term and K^v is called the feed-forward gain. This tracking controller is represented in Figure 2.4.

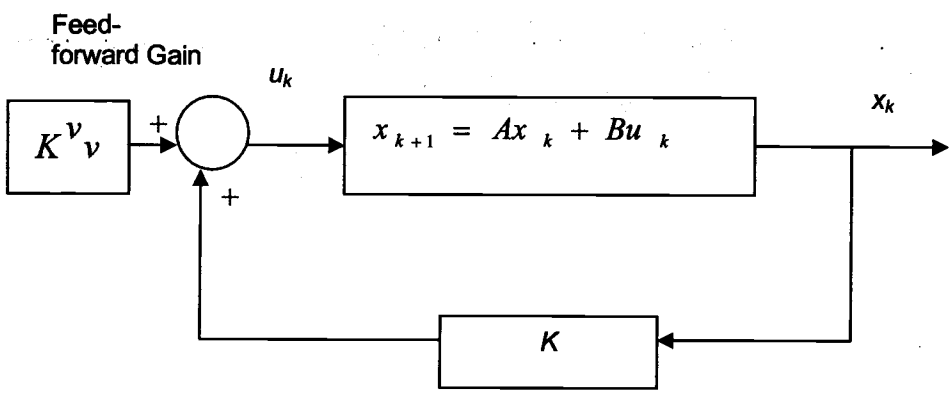


Figure 2.4 LQ tracking controller

Other than the one additional term in the LQ tracking controller, the. LQR and LQ tracking controllers are similar.

2.4 Parameter Estimation and Adaptive Control

As mentioned earlier we have large variations in a system parameter of our mathematical model. In order to take care of this parameter uncertainty, we use parameter estimation in our controller. We

use an average of recent values of the system parameter in our controller. We select an averaging window and estimate our parameter by averaging its actual past values over this window period. We use this estimated parameter value to select the controller gain. Since we adapt our estimate of the parameter, and hence system model used to compute the controller, this type of control is called adaptive control. Adaptive control has many forms. We could use adaptive control if the form of the mathematical model itself changed from time to time, in which case we would switch from one form of a controller to another. The form of our system model does not change, but we do switch between different controller structures. The optimal controller produces negative allocation when the error is positive, so we switch to a different controller structure when error is positive.

Chapter 3

MPEG MEASUREMENTS

Most video applications use the MPEG format to encode and transmit data. MPEG is the most efficient compression algorithm available for video data. MPEG is widely used in digital television set-top boxes, HDTV decoders, DVD players, video conferencing, Internet video and many other applications. The main focus of our project was on CPU scheduling for the decoding and display of real-rate videos that were encoded using MPEG, or SPEG (a scalable version of MPEG, used with intelligent frame dropping), without any *a priori* processing requirement information. In this case, a study of the processing requirements of MPEG gave us useful information about the system we were trying to control. Another variant of the resource allocation problem arises for stored videos. In this case, we can store, transmit, and use information gleaned on the characteristics of the video. For this second problem, we sought to learn whether we could use such information to develop a more effective, or more efficient, scheduler. Both of these considerations prompted us to study the MPEG video streams in detail. We decided to run some tests on MPEG streams to see how their decoding time varied frame-by-frame and whether we could predict the decoding time of the next frame if we knew its characteristics.

In the next section, we present a brief explanation of the MPEG compression algorithm that will aid the reader in understanding our experiments. For more detailed information, refer to [14,15,16]. We describe our experiments and their results in Section 3.2. We propose a possible predictor for frame CPU decoding time based on frame length. In section 3.3, we summarize the key points we learned from our experiments.

3.1 MPEG Fundamentals

MPEG is currently the de-facto standard for compression of video applications. It works well for a wide variety of applications and is standardized. Video compression helps applications because compressed video requires less storage space and also less bandwidth for transmission over the network.

We now explain why video compression is required and how much compression is expected. Consider the HDTV format for broadcasting. This format requires the dimensions of video frames to be 1920 pixels horizontally by 1080 lines vertically and the frame-display rate to be 30 frames per second. Each pixel requires 8 bits to represent the each of three primary colors. Multiplying all these numbers together, we find that the total data rate required to transmit HDTV without compression would be 1.5 Gb/sec. However, each channel is only allocated 6 MHz bandwidth, and so each channel can only support a data rate of 19.2 Mb/sec, of which only 18 Mb/sec is available for video. The remaining bandwidth is needed to support audio and other transport information. These calculations show that we must compress the original video data by a factor of 83:1 to transmit it over such channels with few visual artifacts and little degradation of the original video.

MPEG compression is able to meet the requirements listed above for video compression. Video sequences contain a significant amount of redundant information in both the spatial and temporal directions. The aim of an MPEG encoder is to reduce the level of redundancy in both dimensions as much as possible. The MPEG-1 and MPEG-2 standards are based on "motion-compensated block-based transform" coding techniques, meaning that MPEG compression standards rely mostly on inter-frame correlation. The magnitude of a particular pixel in a frame can be predicted from the pixels of nearby frames. Use of this information results in a non-intra frame coding technique. On the other hand, if the magnitude of pixel were predicted from nearby pixels of same frame, that would be called intra-frame coding. Consider, for example, a single scene in a video that lasts for several frames. In this case, we can assume that much of the detail in the video picture will remain the same for much of the same scene and that there will be a high correlation between adjacent

frames. We can make use of non-intra-frame coding techniques for these situations. However, at the boundaries of scene changes, there will be very little correlation between adjacent frames but there may still be some correlation between nearby pixels within a frame. MPEG video coding is a combination of temporal (non-intra-frame) motion-compensated prediction followed by transform-coding of the remaining spatial (intra-frame) information. We will mainly focus on the non-intra frame coding technique.

In Section 3.1.1, we describe the different layers in MPEG video encoding. In Section 3.1.2, we discuss briefly how intra-frame coding is done. In Section 3.1.3, we explain non-intra-frame coding.

3.1.1 MPEG Video Layers

An MPEG video is broken up into a hierarchy of layers to help with error handling, random search, and editing. The top layer, known as the video-sequence layer, is a self-contained video bitstream. The second layer, known as a group-of-pictures (GoP), consists of one or more intra (I) frames, with only spatial encoding, perhaps along with some P and/or B frames, which use temporal encoding. The functions of the I, P and B frames will be discussed later. The third layer is a picture, which is a single video frame. The fourth and bottom layer is the slice layer, which is a part of the picture or image and is coded independently from other slices of the same picture for error confinement.

Although that completes the formal hierarchy of the layers, there is a further subdivision of the bottom layer. Each slice consists of ordered macroblocks. Macroblocks are 16x16 arrays of luminance (brightness) pixels with 2 8x8 arrays of associated chrominance (color) pixels. Macroblocks are further subdivided into 8x8 blocks for transform coding. This is summarized in Figure 3.1.

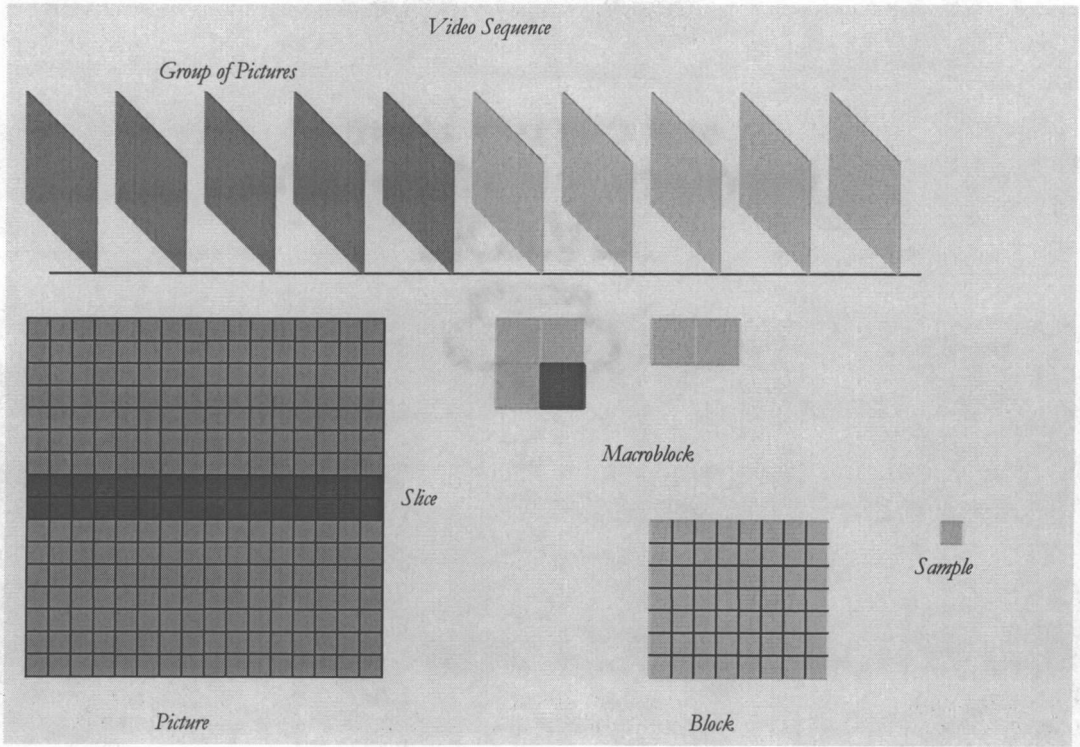


Figure 3.1 MPEG hierarchical structure

3.1.2 Intra-Frame Coding

Intra-frame coding is a compression technique employed to reduce the amount of data required to represent the current frame, with no reference to the adjacent frames. Figure 3.2 shows the block diagram of an MPEG video encoder for intra frames. This is very similar to a JPEG still-image encoder. The basic blocks are the video filter, the discrete cosine transform (DCT), the DCT coefficient quantizer, and the run-length coder.

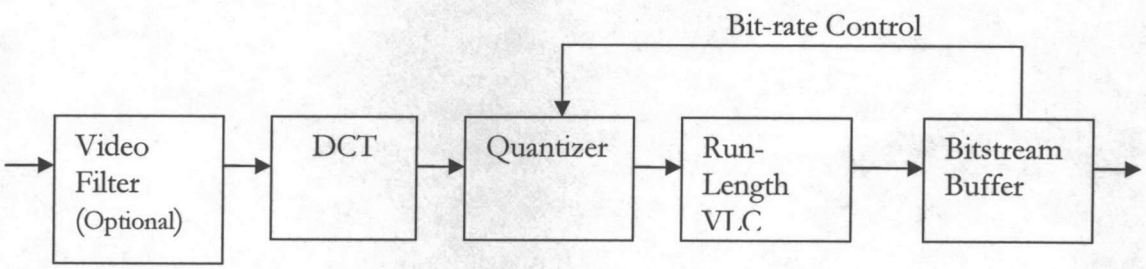


Figure 3.2 Intra-frame encoder

Video Filter

In MPEG, pixels are represented in YCbCr color space rather than red, green and blue color space because the human visual system is less sensitive to changes in chrominance than luminance. Here Y is the luminance signal, Cb is the blue-color difference signal, and Cr is the red-color difference signal. A macroblock can be represented in different formats con, such as 4:4:4, 4:2:2 and 4:2:0. The 4:4:4 format is full-bandwidth YCbCr video with 4 Y blocks, 4 Cb blocks and 4 Cr blocks. The 4:2:2 format contains half as much chrominance information as the 4:4:4 format, and the 4:2:0 format contains one quarter the chrominance information. The 4:2:0 format is sufficient for most consumer-level products. This format allows immediate data reduction of the video from 12 blocks/macroblock to 6 blocks/macroblock. To create this format without generating any artifacts we use a video filter.

Discrete Cosine Transform and Quantization

As mentioned earlier, adjacent pixels within a frame consist of similar data. They tend to be highly correlated. We apply a transform to decorrelate this data. The Discrete Cosine Transform (DCT) is very efficient in doing this. The DCT decomposes the (spatial) signal into its underlying spatial frequencies.

If further increase in data compression is desired, we can then process the DCT transformed image to reduce the precision of the DCT coefficients, further reducing the data required to represent the frame. Since most of the energy in an image is concentrated on edges and high spatial frequencies, the high-frequency coefficients are kept and the low-frequency coefficients are dropped. The removal of the DCT coefficients is controlled on a macroblock-by-macroblock basis, by a quantization block. The goal of this operation is to set as many DCT coefficients as possible to zero, while satisfying the bit-rate and video-quality requirements.

Run-Length Amplitude Coding Using Variable Length Codes

After quantization, we will have a large number of zero coefficients. We can reduce the data required if we can represent this large number of zero coefficients in a more efficient manner. This is the purpose of run-length amplitude coding. Run-length amplitude coding replaces a sequence of zero coefficients and the following non-zero coefficient by a short variable-length code, using a table lookup scheme. The most frequently-occurring sequences are given the shortest codes.

Video Buffer

Since video sequences contain information that varies greatly from picture to picture and even within one picture, the encoded data varies significantly in size from one picture to the next. Thus, the transmission time required to send a frame across a constant-bit-rate channel varies substantially. We buffer the encoded streams before transmitting them to smooth out the bandwidth required for transmission.

3.1.3 Non-Intra Frame Coding Techniques

Intra-frame coding techniques exploited only the spatial redundancy for data compression. Considerably more compression efficiency can be achieved if we also exploit the temporal redundancy in the data. Consecutive frames tend to be very similar in video streams. Inter-frame coding techniques, called block-based motion-compensated prediction, use motion estimation for temporal processing. The full spatially-coded intra-frame is called the *I frame*. This I frame is used as a base for inter-frame coding. The *P frames* and *B frames* derived from the I frames are explained next.

P Frames

From an I frame, the encoder can forward predict a future frame. This forward-predicted frame is known as a P frame. A P frame can also be forward predicted from other P frames. A Group of

Picture (GoP) is defined as a group of frames from one I frame to the frame preceding the next I frame. A GoP that lasts for 5 frames can be represented as I, P, P, P, P, I, P, P, P, ...

Each P frame in this sequence is predicted from the immediately preceding I or P frame.

B Frames

Another possibility is to use both forward and backward motion prediction. This results in *bi-directional* interpolated prediction frames, known as B frames. B frames are coded using forward prediction from the previous I or P frame and backward prediction from the succeeding I or P frame. A GoP consisting of 7 frames including B frames can be represented as I, B, B, P, B, B, P, I, B, B, P... Here, the first set of B frames are predicted from the enclosing I and P frames, and the second set of B frames are predicted from the enclosing P frames.

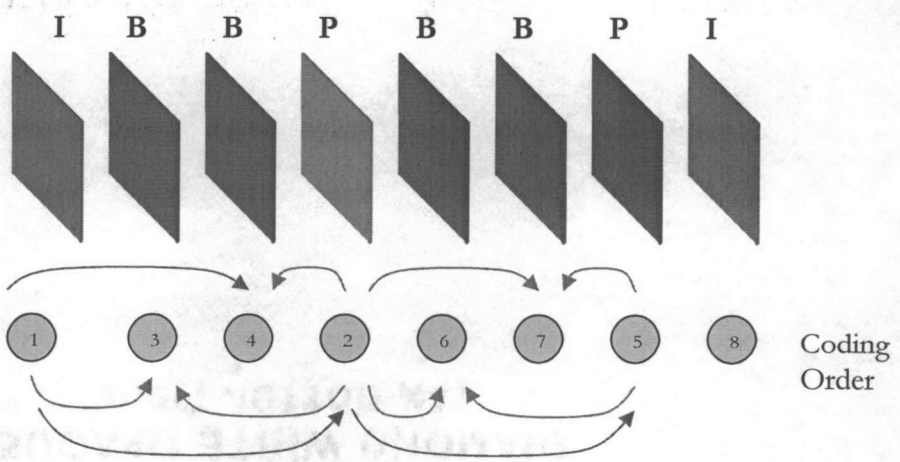


Figure 3.3 Coding order and sending order for GoP video sequence

The coding order and the sending order are illustrated in Figure 3.3. It can be seen form Figure 3.3 that, because of backward prediction, future frames that are to be used for prediction must be encoded and transmitted first. The frame numbers in circles represent the sending (coding) order, and the arrows illustrate the frames that are used to predict other frames. Most video applications

use two consecutive B frames to achieve the ideal trade-off between compression efficiency and video quality.

This was a brief introduction to the MPEG standard. The next section describes the experiments that we performed on the MPEG decoder.

3.2 MPEG Measurements

It is clear from the previous section about the basics of MPEG that MPEG video frames vary greatly in how much data they contain. Decoding times vary greatly between frames, even for adjacent frames. An I frame is self-contained and has the largest decoding time, followed by P and then B frames. When the scene changes, these requirements may also vary from one GoP to another. Because of this variation, it is difficult to reserve CPU for MPEG frame decoding for real-time flows. We would like to find some type of indicator that will help us predict the decoding times of the various frames. If we could find this, we could use information about the frame characteristics to improve our control scheme. We ran some tests to see if we could find a strong (predictive) correlation between frame decoding times and any inherent characteristics of the MPEG frames.

We used a modified Berkeley MPEG player available in the Quasar software package [29] and ran the tests using TSL. We modified the MPEG decoder of this MPEG player to record information on the size and the decoding time of each frame. We used the *gettimeofday( )* routine, available on Unix and Linux systems, to get information on the time when frame decoding started and finished. This routine gives timing information in microsecond resolution. We used several videos to verify our results. We present here the results of these experiments on the video stream described below.

1. Bike.mpg

   - Length: 970 Frames
   - GOP: IBBPBBPBBPBBPBBI...

Figure 3.4 shows the decoding time of sequential frames, and Figure 3.5 shows the size of sequential frames for Bike.mpg.

As we can see from Figure 3.4, the processing times of the frames vary greatly. In this figure, we can also see that frames of the same type have similar processing-time requirements. We observe a jump in the processing requirements after 700 frames that we will ignore for the time being. We concentrate on the frame decoding times of the first 700 frames. Figure 3.6 shows the decoding time of the first 700 frames, zooming in on the *y*-axis scale.

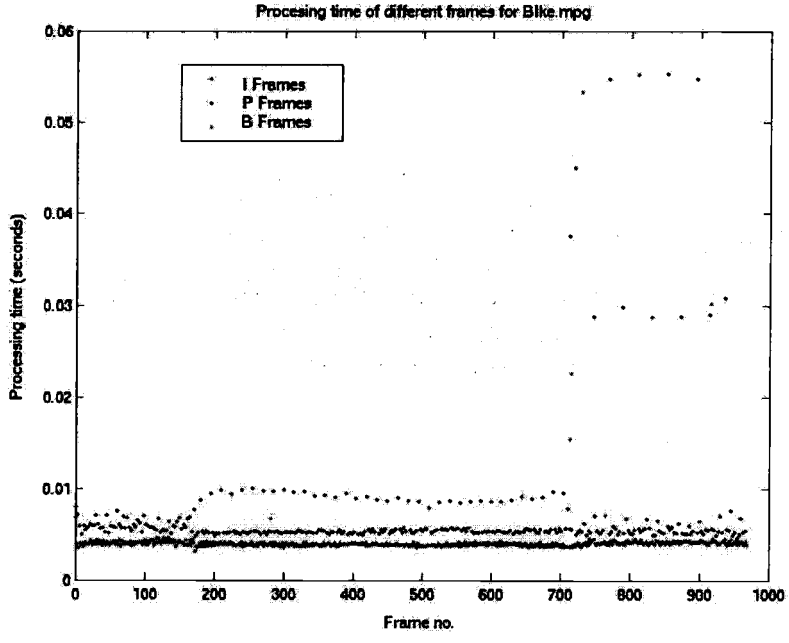


Figure 3.4 Decoding time of sequential frames in Bike.mpg

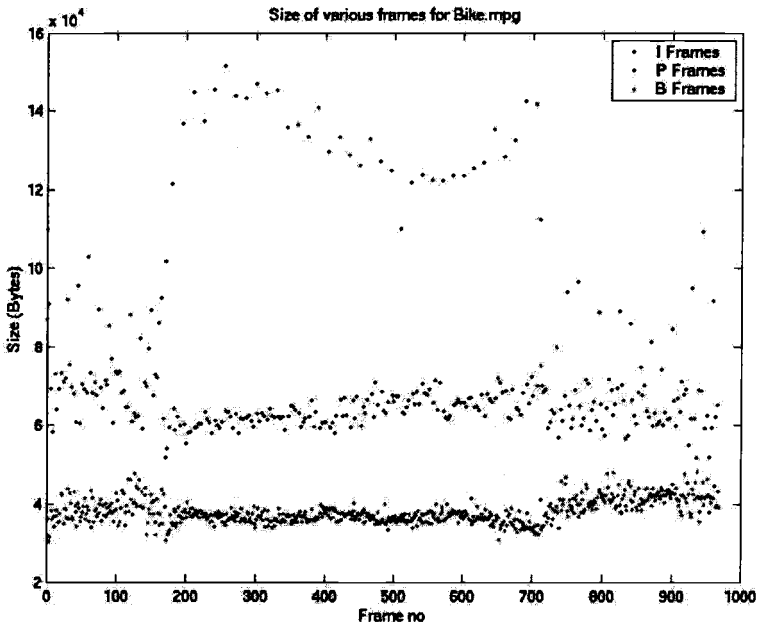


Figure 3.5 Size of sequential frames in Bike.mpg

Figure 3.5 shows the size of the frames in the Bike.mpg video. As discussed earlier, the I frames are the largest in size among the different types of frames since they are self contained and use only intra-frame compression, followed in size by the P and then B frames. There is a striking similarity between the shapes of the plots in Figures 3.5 and 3.6. This gives rise to the hypothesis that the decoding times of frames are directly related to their sizes.

To check this hypothesis, we plot the decoding time of the frames versus their sizes. Figure 3.7 shows this plot.

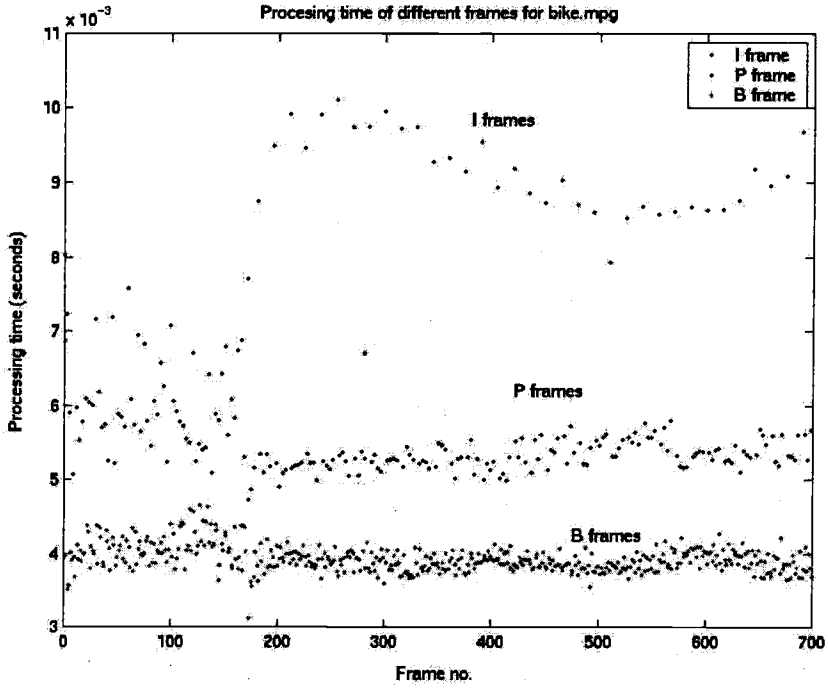


Figure 3.6 Decoding time of the first 700 frames for Bike.mpg

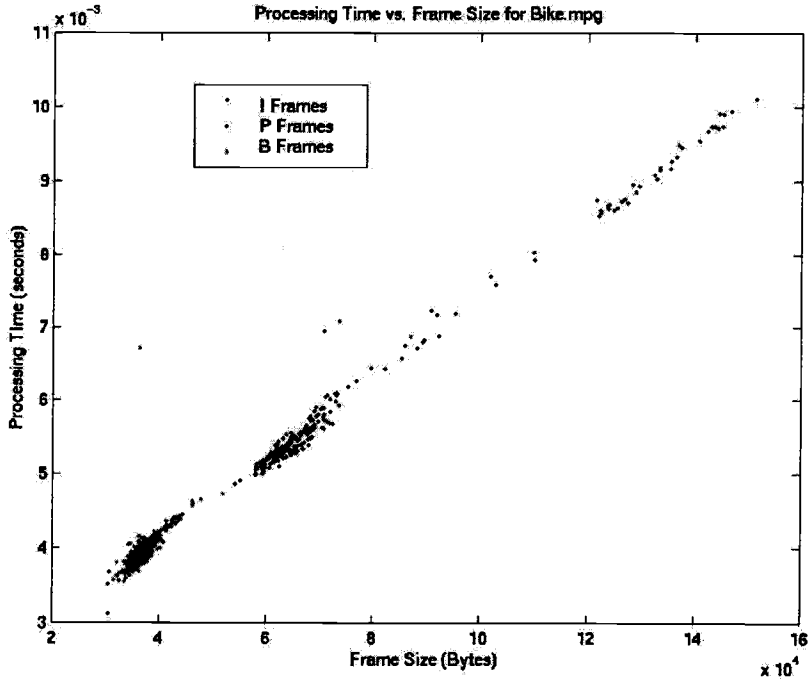


Figure 3.7 Decoding time vs. size of frames for Bike.mpg video

From Figure 3.7, we can see that the decoding time of the various frames indeed has a strong correlation with the frame's size. It is also clear from this figure that the relationship between these variables is linear. (Note here that we have ignored the abnormalities appearing after the first 700 frames.) This result was surprising to us as we did not expect such a profound relationship between decoding time and the size of the frames even though we expected some kind of relationship given the complexity of the MPEG algorithm. Our main goal of running the tests on the MPEG streams was to get a better idea about how the MPEG frame decoding time varied over time. This result can be put to use to improve CPU allocation for streaming stored video, where frame length can be inserted in the frame headers. However, in the case of live real-time videos, we do not have available information about the lengths of the video frames. We are focusing on these types of live real-time videos in this thesis. More importantly, we would also like our CPU scheduler solution to apply to other applications with real-rate flows, not just to video applications. Hence, we will not use this information in the design of our real-rate scheduler. Instead, we use some common information about the structure of MPEG videos that is not specific to any particular video.

Although we will not use the correlation between decoding time and frame size in our real-rate scheduler, we present an example of a decoding-time predictor that can be used for stored videos. Such a predictor was derived earlier by Bavier, et al., at University of Arizona [30]. They developed a predictor based on frame type and frame size.

3.2.1 Possible MPEG decoding time predictor

In this section, we propose a possible MPEG frame decoding-time predictor that can be used for scheduling stored video streams. As is clear from Figure 3.7, the relationship between the decoding time of a MPEG frame and its size is linear. Using this information, we have some confidence that a linear predictor, the simplest form of a predictor, will be adequate. Simple linear curve fitting should work. Figure 3.8 shows a linear curve fit to the data in Figure 3.7.

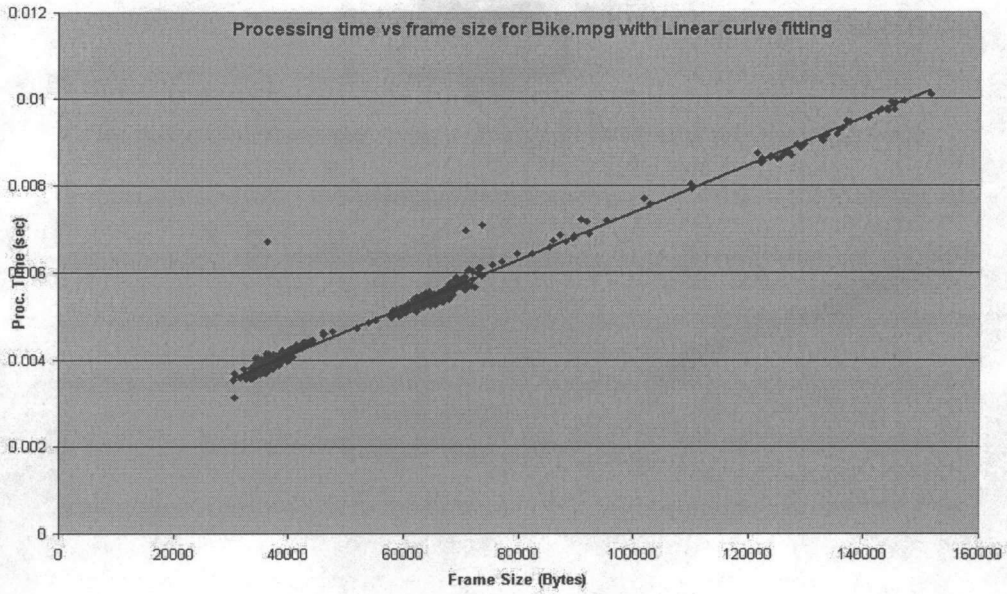


Figure 3.8 Frame processing time vs. size with linear curve fitting

The line is a close fit to the data, as can be seen in Figure 3.8. Linear regression analysis, applied to this curve fitting problem, yields the following predictor for processing time.

$$P = 0.0019 + (5.46 \times 10^{-08})S$$

where P is the predicted processing time of the frame and S is the actual size of the frame. Figure 3.9 shows the percentage deviation of the predicted decoding time from the actual decoding time of each frame.

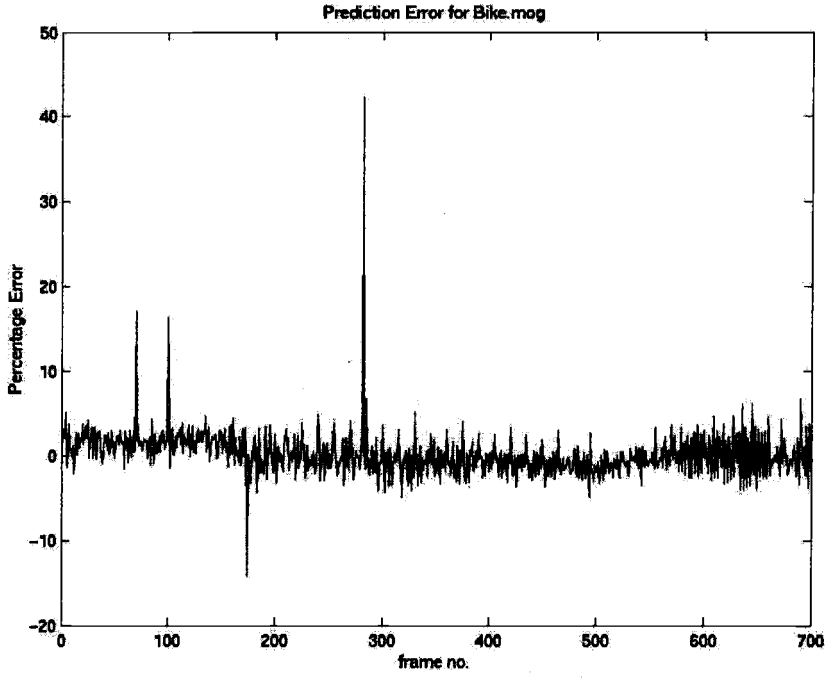


Figure 3.9 Percentage error of decoding time predictor for Bike.mpg

Figure 3.9 shows that our predictor performs quite well. Ignoring the occasional spikes in the plot, we find that our predictor predicts the decoding time of each frame with less than a 5% error. This is a significant result considering the large variation in MPEG frame decoding times. We believe that the spikes are due to the occasional randomness that arises since we do not have complete control of CPU. The CPU may switch to some more important task that has a higher priority. Figure 3.10 shows the absolute error in the predicted decoding time. It shows that we can predict the decoding times of most of the frames with an accuracy of less than 0.5 ms. The maximum deviation of this predicted decoding time is less than 3 ms. As discussed in the first chapter, current general-purpose operating systems have a minimum scheduling granularity of more than 1 ms. Hence, our predictor is quite accurate in predicting the decoding times considering this limitation of current operating systems. In our case, we have scheduling periods of 16 ms in TSL. The time-stamp granularity of the video frame is 33 ms. The predictor performs quite well, with a maximum deviation less than 3 ms..

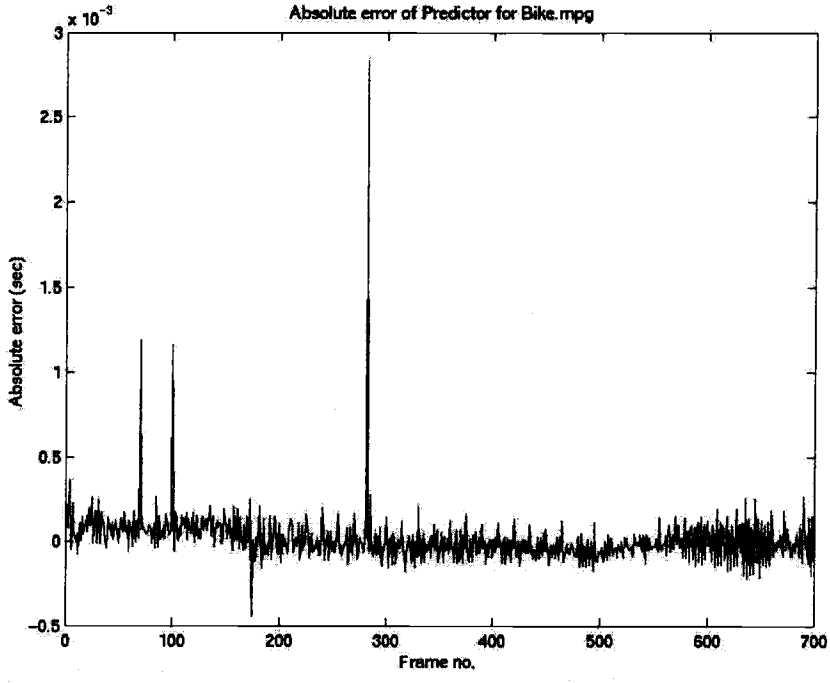


Figure 3.10 Absolute error of predictor for Bike.mpg

One important characteristic of this predictor is that we have not taken into account the frame type in this predictor. We conjectured that we could do even better if we developed a different predictor for each frame type. Applying the same linear curve fitting for each frame type, we obtain the following set of linear predictors.

$$PI = 0.0021 + \left(5.33 \times 10^{-8}\right) SI$$
$$PP = 0.0012 + \left(6.48 \times 10^{-8}\right) SP$$
$$PB = 0.0016 + \left(6.35 \times 10^{-8}\right) SB$$

Where, *PI, PP* and *PB* are the predicted processing times for the I , P and B frames, and *SI, SP* and *SB* are sizes of the I, P and B frames. The percentage error and absolute error of these predictors are shown in Figures 3.11 and 3.12, respectively.

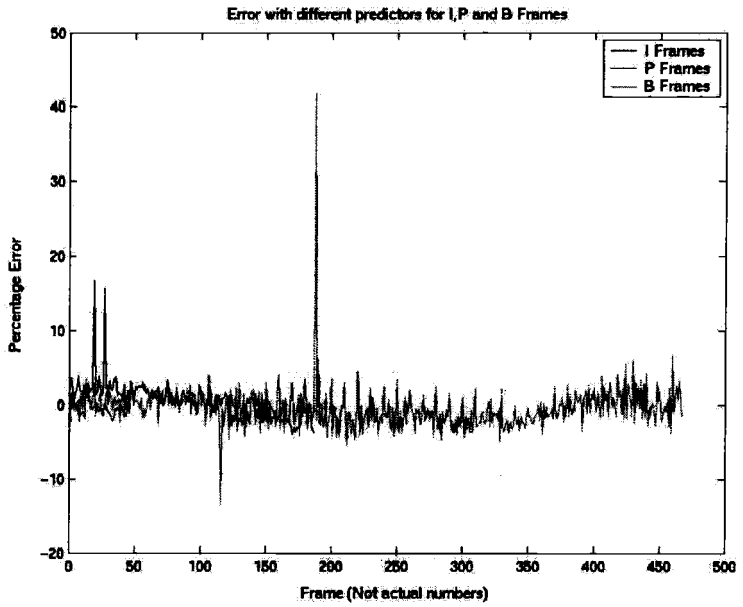


Figure 3.11 Percentage error for each predictor based on frame type

Figures 3.11 and 3.12 show that we do not get any improved performance from including information about the frame type in our predictor. These errors are very similar to the errors encountered in the previous predictor implementation. This contradicts to the results of Bavier et al. They found that performance of predictor improved if information of both frame type and size was used. In our case performance does not improve if frame type information is added to predictor.

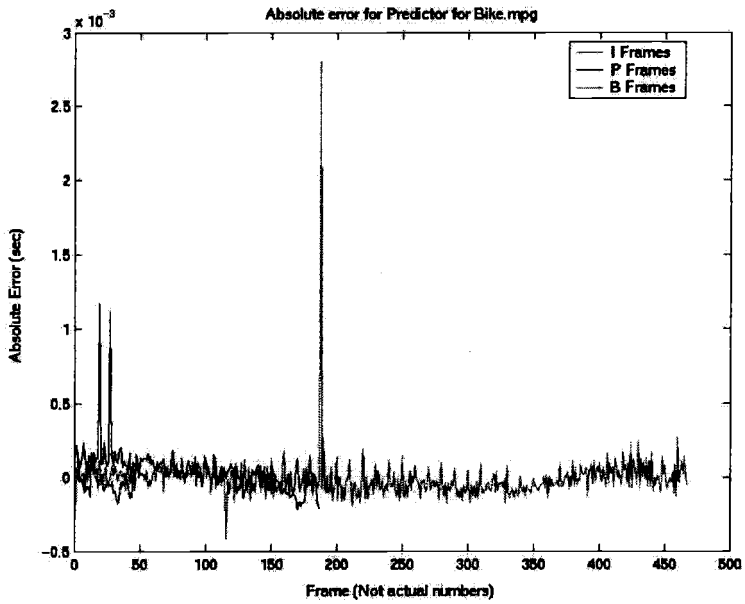


Figure 3.12 Absolute error for each predictor

3.3 Key Points

We have shown by deriving these predictors that it is possible to design improved schedulers for stored videos using information about the frame sizes in the video. This work is preliminary. We have ignored the abnormalities observed after the first 700 frames. We also showed results for only a single video. We would need to repeat these experiments for quite a few other videos in order to be satisfied that these predictors would consistently result in improved performance. If the slope of the linear curve needs to vary for different videos, an adaptive controller could be designed to estimate the predictors based on actual measurements in real time.

Since the main focus of this thesis is on live video streams rather than stored videos, we chose not to explore into this topic further in this thesis. Continued work in this area is left as future research.

We performed other experiments on MPEG video streams that we use in the design of our real-rate scheduler. We postpone the discussion of these other experiments until Chapter 5, following the presentation of our system model in Chapter 4.

Chapter 4

System Overview and Modeling

4.1 Overview of System

In this chapter, we discuss the system to be controlled and the controller structure for that system. Tasks are scheduled in the operating system by the TSL scheduler. This scheduler determines the priority for ordering and the time of execution of each task within a specified time period. The controller determines the length of the execution time, or allocation. Steere, et al., discuss the control issues related to the implementation in some detail [10]. Some modifications have been made to the controller structure, however, since that analysis was done. The scheduler is implemented at a lower level than the controller. The controller can easily be modified to change the control algorithm, which we do in Chapter 5. The controller decides what *proportion* of the CPU to allocate to each task over the specified *period*. We can also alter the *period*, although we have kept it constant. This controller is appropriately called a proportion-period controller.

We have focused on designing a controller to allocate CPU to a task that decodes and displays MPEG video streams. It can be easily adapted to suit other multimedia flows, as well as the processing of other real-rate flows. We break the task up and implement it as multiple threads, with buffers between them where partially-processed video frames can be stored until the next thread can use them. This pipeline-like configuration is shown in Figure 4.1.

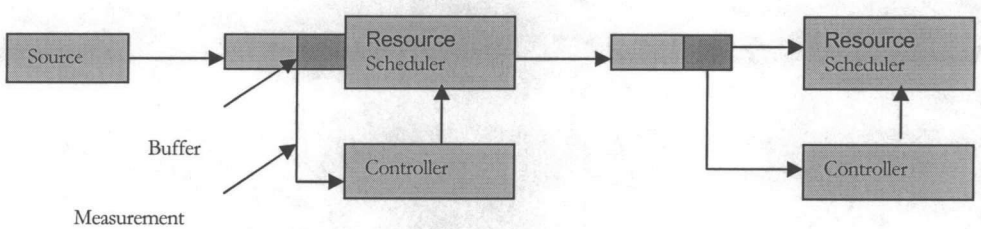


Figure 4.1 Pipeline configuration

At each pipeline stage, a feedback mechanism (controller) decides how much of the resource to allocate to that thread, based on only local measurements. This separates the control of each pipeline stage from that of the others and allows cascading of individual stages without difficulty of implementation. Centralized controllers are difficult to implement in this context since the length and number of pipelines in the system varies dynamically over time. The user may open multiple multimedia displays on the same computer, and other real-rate tasks may be scheduled on the computer.

This is the general pipelined abstraction that can be applied to any system. A more detailed abstraction, depicting the schedulers and controllers specific to our MPEG video application, is shown in Figure 4.2.

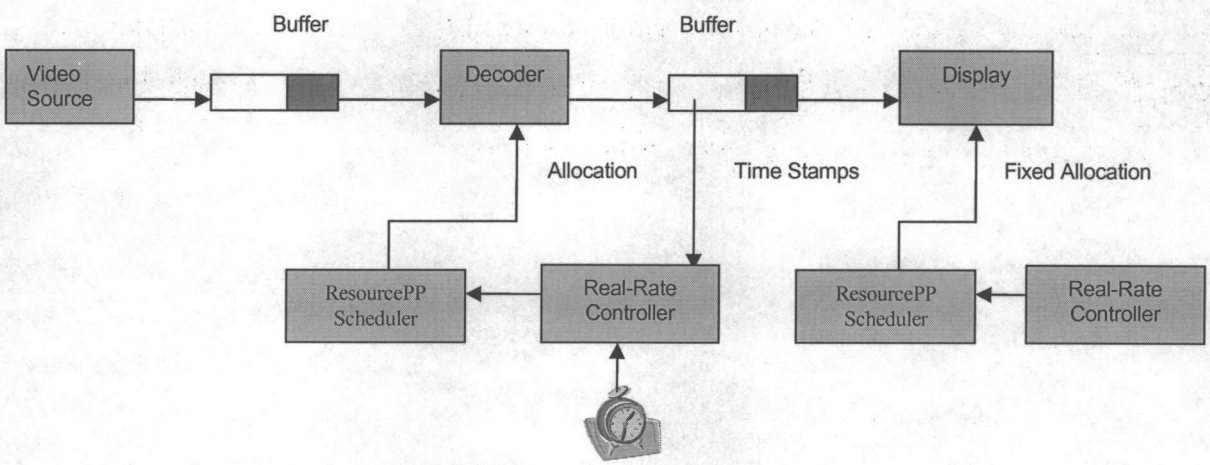


Figure 4.2 Our MPEG video pipeline configuration

As shown in Figure 4.2, we have two pipeline stages in our system. The video source transmits data at a constant rate of 30 frames/sec., and this data is stored, until it can be used, in the input buffer to the first stage of the pipeline. The first stage is the MPEG decoder thread. The decoder takes video frames from the head of its input buffer, decodes them, and places the decoded frames into its output buffer. The CPU allocation to the decoder is decided by the real-rate controller. The

goal of the real-rate controller is to examine the time-stamps of each MPEG frame at the tail of the output buffer and compare them with wall clock time. Based on this comparison, the controller must decide how much CPU to allocate so that the average frame-decoding rate matches the desired rate of 30 frames/sec. The second stage of the pipeline is the video display thread. The display thread takes decoded frames from its input buffer and displays them on the monitor. The allocation required by our display thread is relatively constant, so we have fixed the CPU allocation of the display thread.

The flow along the pipeline must keep up with an externally-driven rate. (Such multimedia flows are called *real-rate flows*.) Poor performance is observed if the MPEG display thread does not have a frame to display available in its input buffer every 33.3 milliseconds. If too many frames were produced ahead of time, to avoid starving the display, the buffer requirements between the decoder and the display would increase. This would also increase the end-to-end latency of the pipeline. Poor performance in the rest of the computer system results if the decoding process is given higher priority over other processes, since decoding is CPU intensive.

To control such a system, we require a measurement that tells how far ahead or behind a flow is from the required real rate or the display's real rate. This can be obtained by marking each frame with a time-stamp that indicates the time offset from the first packet in the flow. These time-stamps represent the application's logical time. For example, this logical time may be the playback time of the video application. The real-rate mechanism's goal is to transmit data in such a way that the real system time stays aligned with these time stamps.

4.2 System Modeling

In this section, we derive the system model and select the error variables that should be kept small.

If t_i is the time-stamp of the packet at the tail of the decoder's output buffer at the current sampling instant i of the controller, and the sampling period is s, then $t_i - t_{i-1}$ would be the logical time-stamp interval between the frames decoded one sampling period apart and the real rate of the flow at

sampling instant i would be $(t_i - t_{i-1})/s$. We define the first error variable, z^1, to be the difference between the current real rate and the target real rate of one.

We also need a second error variable, z^2, to keep track of the long-term deviation of the real rate from the desired real rate. We simply add the individual real rates at each sampling instant. This should equal the total real-time progress made up to the current time. The error variable is the difference between the total real-time progress and the desired real-time progress (i). We wish to minimize these two error variables.

$$
\begin{aligned}
z_i^1 &= (1/s)(t_i - t_{i-1}) - 1 \\
z_i^2 &= \sum_{n=1}^{i} (1/s)(t_n - t_{n-1}) - 1 \\
\Rightarrow z_i^2 &= (1/s)t_i - i
\end{aligned}
\qquad (4.1)
$$

In order to design a controller using analytical methods, we must describe the system's dynamic behavior using a control-oriented mathematical model. The following model describes our system reasonably well, except that it omits the granularity of time-stamps.

$$t_i = t_{i-1} + (1/k_i)p_{i-1} - (1/k_i)n_{i-1} \qquad (4.2)$$

Here, p_{i-1} is the allocation assigned during the last period, n_{i-1} is the amount of assigned allocation not used by the application in the last period, and k_i is the variable that relates the amount of progress made to the allocation actually used by the application. The variable k_i is random by nature and varies by a factor of four, as we observed in the experiments in Chapter 3.

Equation 4.2 can be rearranged as

$$
\begin{aligned}
t_i - t_{i-1} &= (1/k_i)(p_{i-1} - n_{i-1}) \\
\Rightarrow k_i &= (p_{i-1} - n_{i-1})/(t_i - t_{i-1})
\end{aligned}
\qquad (4.3)
$$

From Equation 4.3 we find that k_i relates the amount of progress made in terms of time-stamps to the cycles actually run by the system. We know from MPEG measurements that the processing

times of MPEG frames varies by a factor of four, so we expect k_i to vary by a factor of four as well. We will discuss the quantization issues in the Chapter 5.

We now have both the system model and the error variables that we want to minimize. In the next section, we set up the minimization problem as an optimal tracking problem.

4.2.1 Control Design Set Up as a Dynamic Optimization Tracking Problem

We have found a state-space system model for the system (4.2) and determined which errors we want to minimize (4.1). The first error variable z^1 includes the previous state t_{i-1} . Therefore, we must augment the state space so that we can obtain expressions for the error variables in terms of only current state variables. We select the augmented states to be $x_i^1 = z_i^1 + 1$ and $x_i^2 = z_i^2 + i$. We now minimize x_i^1 - 1and x_i^2 - i, forcing the new state variables (x_i^1, x_i^2) to track (1, i). This is the same as minimizing the original error variables (z_i^1, z_i^2). The new state equations are given in Equation 4.4.

$$x_i^1 = z_i^1 + 1 = (1/s)(t_i - t_{i-1}) = (1/(k_i s))p_{i-1} - (1/(k_i s))n_{i-1}$$

$$x_i^2 = z_i^2 + i = (1/s)t_i - i + i = (1/s)t_i$$

$$\Rightarrow x_i^2 = \left((1/s)t_{i-1} + (1/(k_i s))p_{i-1} - (1/(k_i s))n_{i-1}\right) \qquad \text{[From Equation (4.2)]}$$

$$\Rightarrow x_i^2 = x_{i-1}^2 + (1/(k_i s))p_{i-1} - (1/(k_i s))n_{i-1}$$

Our augmented state equations now are given by

$$x_i^1 = (1/(k_i s))p_{i-1} - (1/(k_i s))n_{i-1}$$

$$x_i^2 = x_{i-1}^2 + (1/(k_i s))p_{i-1} - (1/(k_i s))n_{i-1}$$

Also, $p_{i-1} - n_{i-1}$, the number of cycles run in the last period, is directly measurable from the system. The controlled variable p_{i-1} dominates, and the noise n_{i-1} *is* relatively very small, so we neglect n_{i-1} in our control design. Taking $u_{i-1} = p_{i-1}$, we write the state equations in matrix form:

$$\begin{bmatrix} x_i^1 \\ x_i^2 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{i-1}^1 \\ x_{i-1}^2 \end{bmatrix} + \begin{bmatrix} 1/(k_i s) \\ 1/(k_i s) \end{bmatrix} u_{i-1} \qquad (4.5)$$

The next step is to select a cost functional to be minimized. In our case, we want the augmented states to track unity. In addition, we want to minimize the CPU resource allocated to avoid over-allocating to the application, so we include the input variable in the cost functional. Incorporating these considerations, the cost functional is given in the equation below:

$$J = \lim_{N \to \infty} \left\{ \sum_{i=0}^{N} ((x_i - \bar{x})' Q_i (x_i - \bar{x}) + u_i' R_i u_i) \right\} \qquad \text{where, } \bar{x} = \begin{bmatrix} 1 \\ i \end{bmatrix} \quad (4.6)$$

Here, *x* is the state vector, *u* the input allocation and *i* the index of the sampling period. The matrices *Q* and *R* are the weightings given to the state and input variables. They are positive semi-definite and positive definite matrices, respectively. When designing and testing the controller on our particular problem, we selected *Q* to be the identity matrix, and weighted the scalar *R* to be 1/1000th of Q's weighting of one. It is possible to weight the first and second state variables differently, but we chose to weight them the same. The controller solution equations below are in their general form and apply to different choices of *Q* and *R*.

Selecting a controller to minimize the cost functional (4.6), with the constraint that the state must obey the augmented state equation (4.5), we find that we should use the allocation given in equation (4.7), which is the steady-state solution to the standard LQR tracking problem with the augmented states. The steady-state solution exists if the matrix pair (A, B) is completely controllable. This condition is satisfied in our case. The solution is

$$u(x_i) = Kx_i + K^v v$$

Where, $K = -(B'SB + R)^{-1} B'SA$

$S = A'[S - SB(B'SB + R)^{-1} B'S]A + Q$ (4.7)

$K^v = (B'SB + R)^{-1} B'$

$v = (A - BK)'v + Q\bar{x}$

The matrix K is the feedback gain for our system, and K^v is the feed-forward gain. The positive definite matrix S is the solution to the Riccati equation. We must solve this Riccati equation in order to calculate the gain. Since we want to re-solve the problem on-line for a new controller when our estimate of the k_i parameter changes, we need a numerical solution that requires little computational overhead.

We found a solution to the Riccati equation analytically using a generalized eigenvalue problem [11,12] and derived equations to compute the gain matrices in terms of easily-computed variables. The next section describes how we solved this Riccati equation analytically for our problem. Section 4.2.3 summarizes the steps we implemented on-line to calculate the required CPU allocation.

4.2.2 Solution of Algebraic Riccati Equation

The generalized Algebraic Riccati Equation is

$$F'XF - X - F'XG_1(G_2 + G_1'XG_1)^{-1}G_1'XF + H = 0$$

Comparing it with our Riccati equation,

$$F = A, \quad G_1 = B, \quad G_2 = R, \quad \text{and} \quad H = Q$$

If we define a new matrix $G = G_1 G_2^{-1} G_1'$, then we can express G as

$$G = (1/R)\begin{bmatrix} 1/c \\ 1/c \end{bmatrix}\begin{bmatrix} 1/c & 1/c \end{bmatrix} = (1/R)\begin{bmatrix} 1/c^2 & 1/c^2 \\ 1/c^2 & 1/c^2 \end{bmatrix}$$

where $c = k_i s$, since R is a scalar in our problem. To set up the generalized eigenvalue problem, let

$$M = \begin{bmatrix} F & 0 \\ -H & I \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

$$L = \begin{bmatrix} I & G \\ 0 & F' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1/(c^2R) & 1/(c^2R) \\ 0 & 1 & 1/(c^2R) & 1/(c^2R) \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We find the generalized eigenvalues as follows:

$$M - \lambda L = \begin{bmatrix} -\lambda & 0 & -\lambda/c_1 & -\lambda/c_1 \\ 0 & 1-\lambda & -\lambda/c_1 & -\lambda/c_1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1-\lambda \end{bmatrix}$$

$$|M - \lambda L| = (-\lambda)\begin{vmatrix} 1-\lambda & -\lambda/c_1 & -\lambda/c_1 \\ 0 & 1 & 0 \\ -1 & 0 & 1-\lambda \end{vmatrix} + (-\lambda/c_1)\begin{vmatrix} 0 & 1-\lambda & -\lambda/c_1 \\ -1 & 0 & 0 \\ 0 & -1 & 1-\lambda \end{vmatrix} - (-\lambda/c_1)\begin{vmatrix} 0 & 1-\lambda & -\lambda/c_1 \\ -1 & 0 & 1 \\ 0 & -1 & 0 \end{vmatrix}$$

Solving $|M - \lambda L| = 0$, we solve the resulting equation, $(1+1/c1)\lambda^3 - (2+3/c1)\lambda^2 + (1+1/c1)\lambda = 0$, to find the following generalized eigenvalues:

$$\lambda = \frac{(2c_1 + 3) \pm \sqrt{4c_1 + 5}}{2(c_1 + 1)}, 0$$

where $c_1 = R(k_i s)^2$.

Now computing the eigenvectors, we set

$$\begin{bmatrix} -\lambda & 0 & -\lambda/c_1 & -\lambda/c_1 \\ 0 & 1-\lambda & -\lambda/c_1 & -\lambda/c_1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1-\lambda \end{bmatrix} \begin{bmatrix} e1 \\ e1 \\ e3 \\ e4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Solving this, we get

$$e1 = e3,\ e2 = -\left(\frac{-\lambda}{1-\lambda}\right)e1,\ e4 = \left(\frac{-\lambda}{(1-\lambda)^2}\right)e1$$

So, for $\lambda = 0$,

$$\text{eigenvector1} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

For other stable λ,

$$\text{eigenvector2} = \begin{bmatrix} 1 \\ e2 \\ 1 \\ e4 \end{bmatrix}$$

So,

$$U = \begin{bmatrix} 1 & 1 \\ 0 & e2 \\ 1 & 1 \\ 0 & e4 \end{bmatrix}$$

$$\Rightarrow U_1 = \begin{bmatrix} 1 & 1 \\ 0 & e2 \end{bmatrix},\ U_2 = \begin{bmatrix} 1 & 1 \\ 0 & e4 \end{bmatrix}$$

$$S = U_2 U_1^{-1} = \begin{bmatrix} 1 & 1 \\ 0 & e4 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & e2 \end{bmatrix}^{-1}$$

$$\Rightarrow S = \begin{bmatrix} 1 & 0 \\ 0 & e4/e2 \end{bmatrix}$$

This is the solution of the Riccati equation.

Hence, feedback gain K for our controller would be:

$$K = -(B'SB + R)^{-1} B'SA$$

Computing the *K*,

$$B'SB = \begin{bmatrix} 1/c & 1/c \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & e4/e2 \end{bmatrix} \begin{bmatrix} 1/c \\ 1/c \end{bmatrix}$$

$$\Rightarrow B'SB = 1/c^2(1 + e4/e2)$$

$$B'SA = \begin{bmatrix} 1/c & 1/c \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & e4/e2 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\Rightarrow B'SA = \begin{bmatrix} 0 & 1/c(e4/e2) \end{bmatrix}$$

$$K = -((1/c^2)(1+e4/e2)+R)^{-1}\begin{bmatrix} 0 & (1/c)(e4/e2) \end{bmatrix}$$

Computing feed forward gain K^v,

$$K^v = (B'SB + R)^{-1} B'$$

$$K^v = ((1/c^2)(1+e4/e2)+R)^{-1}\begin{bmatrix} 1/c & 1/c \end{bmatrix}$$

$$v = \begin{bmatrix} 1 \\ -1 + (i * c^2)/(k * (e4/e2)) \end{bmatrix}$$

$$k = -((1/c^2)(1+e4/e2)+R)^{-1}$$

So, now the input will be:

$$u = Kx_{i-1} + K^v v$$

4.2.3 Implementation steps for tracking controller

The steps to calculate allocation online would be:

1. Calculate eigenvalue:

$$\lambda = \frac{(2c_1 + 3) - \sqrt{4c_1 + 5}}{2 * (c_1 + 1)}$$

where, $c_1 = R * (k_i s)$

2. Calculate eigenvector:

$$e1 = e3 = 1$$

$$e2 = -\left(\frac{-\lambda}{1-\lambda}\right) e1$$

$$e4 = \left(\frac{-\lambda}{(1-\lambda)^2}\right) e1$$

3. Calculate feedback and feed forward gains:

$$K = -((1/c^2)(1 + e4/e2) + R)^{-1} \begin{bmatrix} 0 & (1/c)(e4/e2) \end{bmatrix}$$

$$K^v = ((1/c^2)(1 + e4/e2) + R)^{-1} \begin{bmatrix} 1/c & 1/c \end{bmatrix}$$

4. And, finally, calculate, input *u.*

$$u = K * x_{i-1} + K^v v$$

Statistics collected from running the system are used to estimate the average k_i in the state equation, and the estimated value of k_i is used to compute the controller. We will use sliding window averaging online to estimate k_i. One important question that needs to be addressed is, what would be the optimal length of this sliding window? Our MPEG measurements will help us choose this window length. In next chapter, we discuss how we choose this window length based on MPEG measurements. Then, we discuss the implementation results of this optimal controller. We encountered several problems while implementing this controller. We added several features to this basic optimal controller part in order to alleviate these problems. As a result we have several

implementations of our controller. We discuss all these implementations and their results in the next chapter.

Chapter 5

Implementation and Results

In this chapter, we will explain the results we obtained for the MPEG video resource allocation problem with the various controller implementations that we developed. We explain how our controller progressed from a basic optimal controller to a more complex switching controller that switches between the optimal part and an estimating part with several averaging features. Since the optimal control part will have negative allocation when the error is positive, we need a different strategy to calculate allocation in this case. We estimate the desired allocation based on past measurements when the error is positive and when the error is close to zero. Averaging the parameter k over an averaging window smoothes our estimate for *k*. This is the basis of the various implementations. Section 5.1 explains how different length averaging windows will smooth out the varying parameter 1/k in our system. Section 5.2 explains various implementations of the controller and the advantages and disadvantages of each. Section 5.3 describes our conclusion about the performance of the controller.

5.1 Averaging 1/k

We know from previous experiments that the processing times of various MPEG frames vary greatly, in fact by a factor of four. As a result, *k* will also vary by a factor of four, as explained in a previous chapter. One important decision we have to make is which value of *k* to use in developing our controller, which is an optimal controller when the value of k is a constant and correctly models the system.

There are a number of "obvious" solutions to this problem, each of which for various reasons does not solve the problem satisfactorily. One simple solution is to use the value obtained from the preceding frame – the one just decoded. This, however, is not the desired solution for an MPEG video stream. Recall from Chapter 3 that, when the frame transition is from a complete intra frame (I

frame) to a predicted frame (P or B frame), the decoding time will have a definite jump, as can be seen in Figure 5.1, which was also presented in Chapter 3. Therefore, we cannot use the previous frame's processing time as an estimate of the decoding time for the next frame. If we do use it to design a controller, we can be sure that the controller will not be optimal at many sampling instants. Another possible solution is to use the average value of k over the entire video to design the controller. This solution might work for stored videos if we had information on the frame decoding times for the entire video beforehand. Unfortunately, the decoding times of MPEG movies are not typically measured and encoded with the movie, so they may not be available even for stored videos. The decoding times may also vary in different ways for different computer configurations. Also, for live videos, there is no way to obtain this information *a priori*. A third possible solution would be to predict the decoding time based on the frame type. The solution that we use eventually is somewhat related to this observation, although it is not a direct use of frame type. Our goal was to design a general approach to resource allocation for problems with real-rate flows, not to design an implementation that would work only for MPEG video processing.

We conjecture that the decoding times of the frames will have some average value that will not vary too rapidly if averaged over some reasonable length of the video. We propose to measure the average decoding time over an appropriate number of the most recently-decoded frames and to use that value to update our CPU resource controller. We call this approach "sliding window averaging". We slide our averaging window of length n forward and update the controller gains every n frames. The best value for n must be determined.

The first step is to select the optimal length of this window. We need to choose the window length in such a way that the window will always contain a mix of all types of frames. Our intuition was to use a Group of Pictures (GoP) as the window length. We ran some tests on an MPEG video clip to see how different lengths of this window smooth out the variations in CPU processing time. We expected that when the window length was around the length of a GOP for a video that most of the variations should smooth out. The reason for this hypothesis was that in a length of GoP, we will have a good mix of all the frame types (I, B and P), and in fact the same mix for each GoP. This will

take into account the processing time needs of all types of frames. We also know from earlier experiments that, for a given video, frames of the same type have similar processing time requirements. This is shown in Figure 5.1, which was presented in Chapter 3 and is shown here again for convenience.

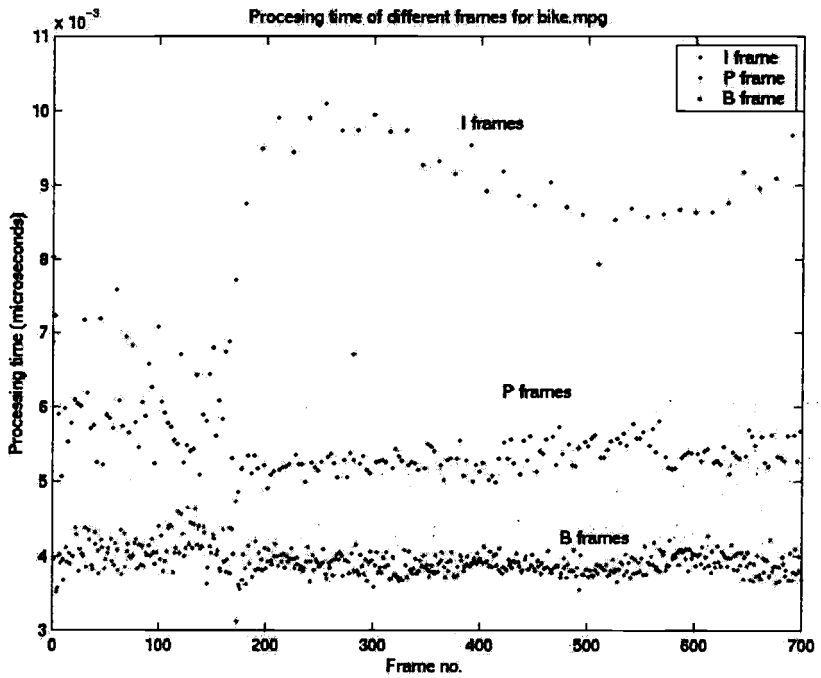


Figure 5.1 Processing times of different frames in Bike.mpg labeled by frame type.

We repeat Equations 4.2 and 4.3 for convenience:

$$t_i = t_{i-1} + (1/k_i)p_{i-1} - (1/k_i)n_{i-1} \qquad (5.1)$$

and

$$t - t_{i-1} = (1/k_i)(p_{i-1i} - n_{i-1})$$

Hence

$$\frac{1}{k_i} = \frac{t_i - t_{i-1}}{p_{i-1} - n_{i-1}}$$

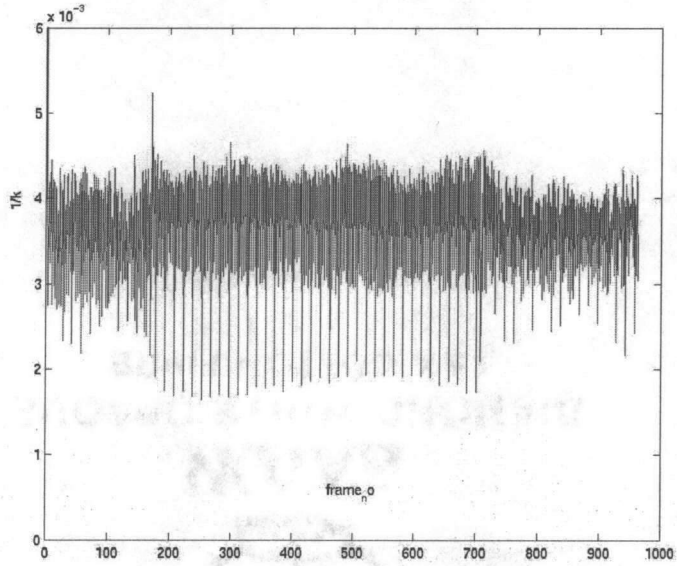
Here, p_{i-1} is the allocation assigned during the last period, n_{i-1} is the amount of assigned allocation not used by the application in the last period, and k_i is the variable that relates the amount of progress made to the allocation actually used by the application.

The progress per CPU cycle is therefore $1/k_i \approx \frac{t_i - t_{i-1}}{p_{i-1}}$, neglecting n_{i-1}.

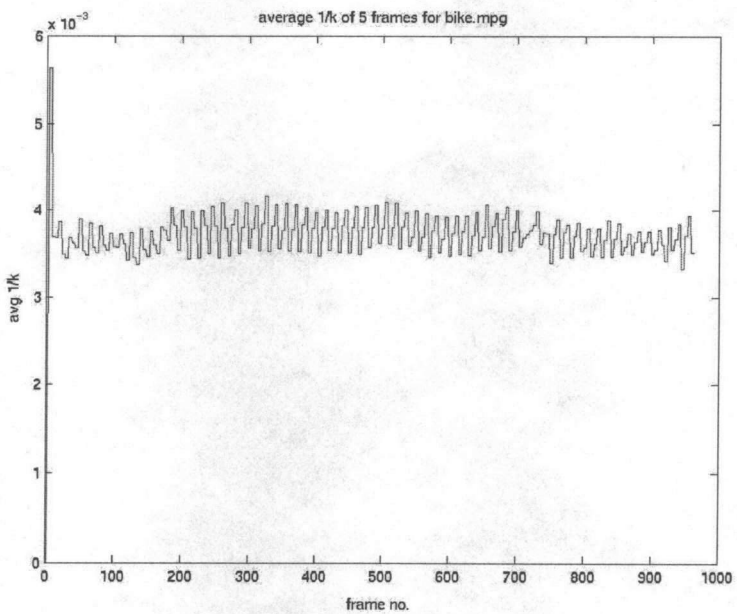
The time-stamps of sequential MPEG video frames are marked 33.3 ms apart, since the frame rate is 30 frames per second. Thus, when we measure the t_i in the MPEG decoder, it is always a multiple of 33.3 ms. Thus, $t_i - t_{i-1}$ is always a multiple of 33.3 ms: 0, 33.3 ms, 66.6 ms, etc. Which multiple depends on whether, and how many, frames complete their decoding within a particular sampling period. This affects the accuracy of our measurements of $1/k_i$.

Figure 5.2 shows the average of $1/k_i$ when using different averaging window lengths.

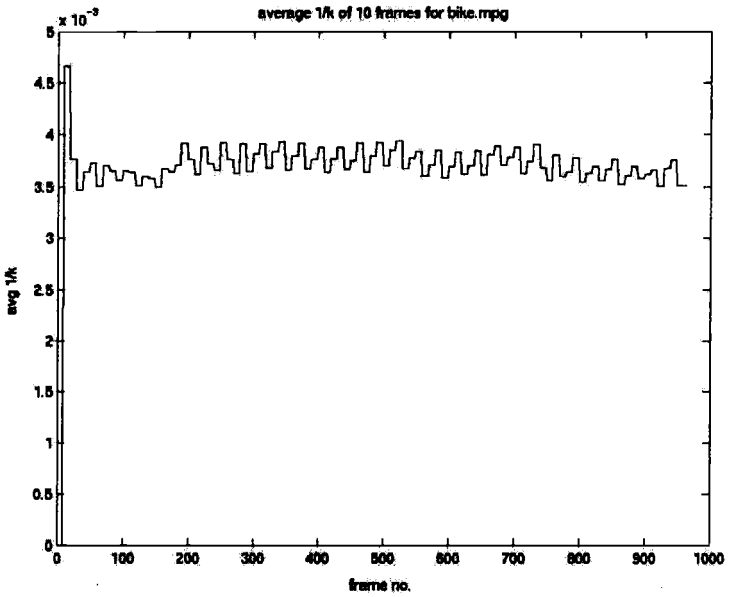
Figure 5.2 Plots of progress per CPU cycle (1/k) for different length windows for the Bike.mpg video



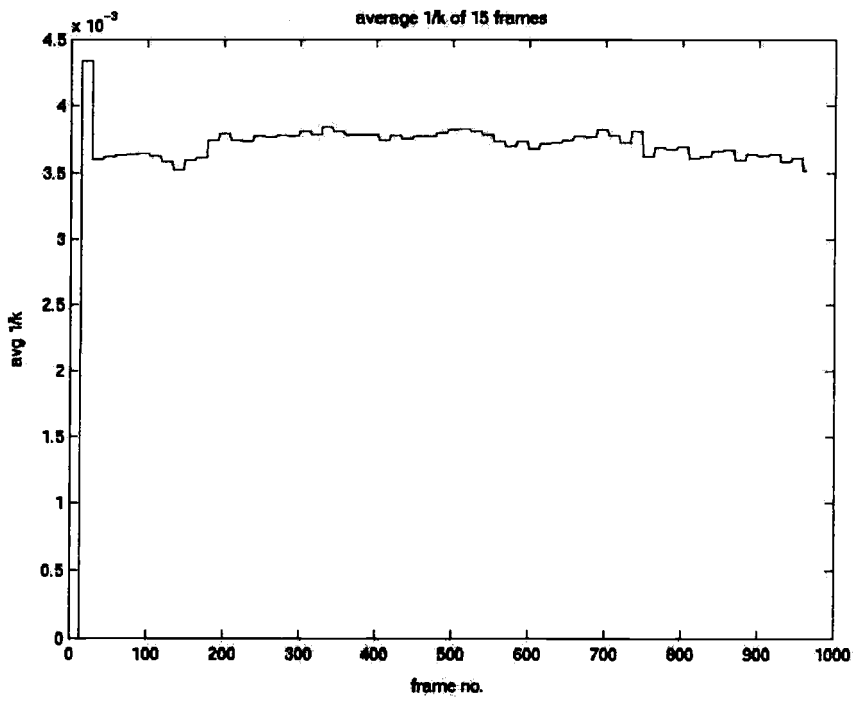
(i) No averaging



(ii) Average of 5 frames



(iii) Average of 10 frames



(iv) Average of 15 frames (GOP Length)

As can be seen from the plots, when the averaging window is 15 frames (one GoP), most of the variations in *1/k* are smoothed out. This result is consistent with our expectation that variations in the frame processing times are all but smoothed out when averaging over the length of a GOP. Based on these results, we decided that an averaging window of 15 frames (GOP length) would be a good choice for our controller.

5.2 Controller Implementation

We implemented the controller described in Section 4.2.3 on a TSL kernel scheduler with Red Hat Linux 7.3. The system that we used had a 2.0 GHz Pentium IV with 1 GB of memory. In our system, negative long-term error means that the video decoding is lagging behind its real rate and positive long-term error means that the video is ahead. This is clear from Equation (5.2), given below.

$$z_i^2 = (1/s)t_i - i \qquad (5.2)$$

In the next section we will discuss the quantization issues inherent in our system. These quantization arise from the different granularities of the time-stamps, the period of scheduler and the period of the optimal controller.

5.2.1 Quantization issues

There are two sources that will contribute to quantization in our system. First, we have a 33.3 ms. granularity in the time-stamps of the video frames. We also have the different sampling periods of the scheduler and the controller.

Granularity of Time-stamps

Granularity of application time-stamps plays an important role in designing the controller. This is due to the fact that the controller cannot measure the progress of the application at a finer granularity than the time-stamp granularity.

Consider the MPEG video application on which we will test our controller. The MPEG video frames are encoded at a clock rate of 30 frames/second. Adjacent frames are marked 33.3 ms. Apart. Each time a frame is decoded between consecutive sampling instances, the measured time stamp t_i increases by 33.3 ms. Blocks in each frame have same time-stamp as the overall frame. Due to this, we can not get any information about the progress of the application at a finer granularity than 33.3 milliseconds. Thus, sampling and updating the controller more frequently than 33.3 ms. will not improve the accuracy of the control. The granularity of the measurements is also one of the reasons that we decided to use averaging. For small errors, the estimating control described in section 5.2.4 results in less overreaction to the granularity than a controller based on the measured error. This will be discussed in more detail later.

Sampling Period of Scheduler

The sampling period of the scheduler in TSL, implemented by Goel, et al., is 16 ms. The controller updates the allocation to the video decoder application every 16 milliseconds. Even if the frame is decoded sometime within this sampling period and new information is available, the controller only updates the allocation at the end of current scheduling period when the new period is about to start. Goel, et al., made this design choice at some point because, while running some tests, they found that reducing the sampling period below 10 ms does not improve the performance any further. In fact, performance may start to suffer. We used the 16 ms period, since we conducted the tests while that was the default scheduling period in TSL. (Goel has since changed that value.)

In summary, measurements are available every 16 milliseconds when the controller is run, but the values are only multiples of 33.3 milliseconds of application time. The granularity of the time-stamps is larger than the period of the scheduler of the system. It is useless to update the allocation any more frequently than the application time-stamp granularity since progress information is inaccurate within that time frame. The error would also appear to vary as actual time progressed while a single frame was being decoded, and the measured error would not coincide with the actual progress that the decoder is making in decoding the frame. Due to this, the measured error will vary within some

range. This can be viewed as measurement noise. If used to compute allocation, it would introduce a disturbance in the system input. The magnitude of these variations in the measured error is determined by the relationship between the 33.3-millisecond time-stamp granularity and the 16 millisecond sampling period. We will present this relationship later in this section. Ideally, we should like the sampling period to be a multiple of the time-stamp granularity. This condition is not satisfied in our case since 16 millisecond is not a multiple of 33.3 milliseconds. The thread's time-stamp period and the sampling period do not coincide. The consequences, for the quantization in our error, are shown in Figure 5.3.

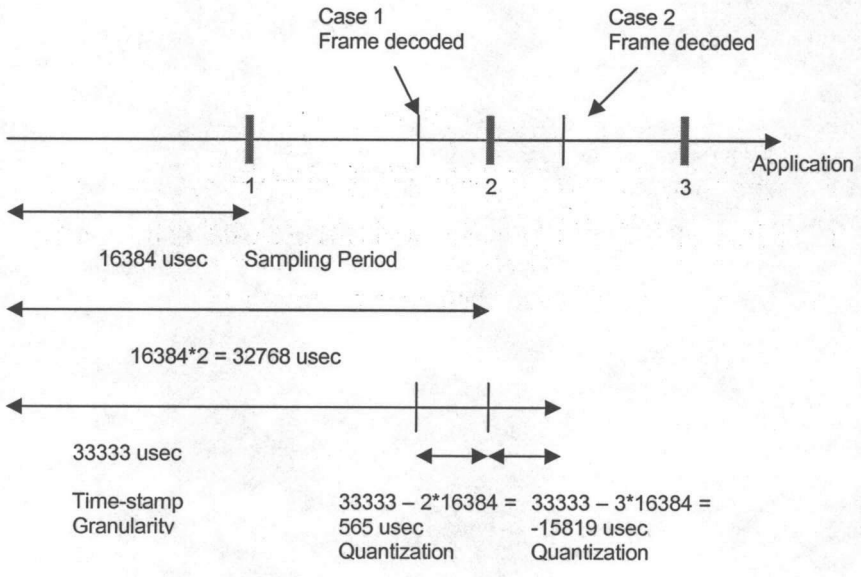


Figure 5.3 Quantizations in Measured Error

As shown in Figure 5.3, we have two possible situations. The video frame might be decoded before two sampling periods, in which case we will observe a positive error. In the second case, the video frame may be decoded after two sampling periods, in which case we will observe a negative error. The equation for this error quantization is given below:

$$e = 33333 - n * 16384$$

where n = number of sampling periods, e = quantization in error.

For n = 1 and 2 we will have a positive error; for n larger than 2 we will have a negative error. This is also the quantization in our long-term error: the long-term error will be multiple of this quantization error e.

The most common cases are shown in Figure 5.3. When the frame is decoded within 2 or 3 sampling periods, we will have an error of 565 or -15819 microseconds, respectively.

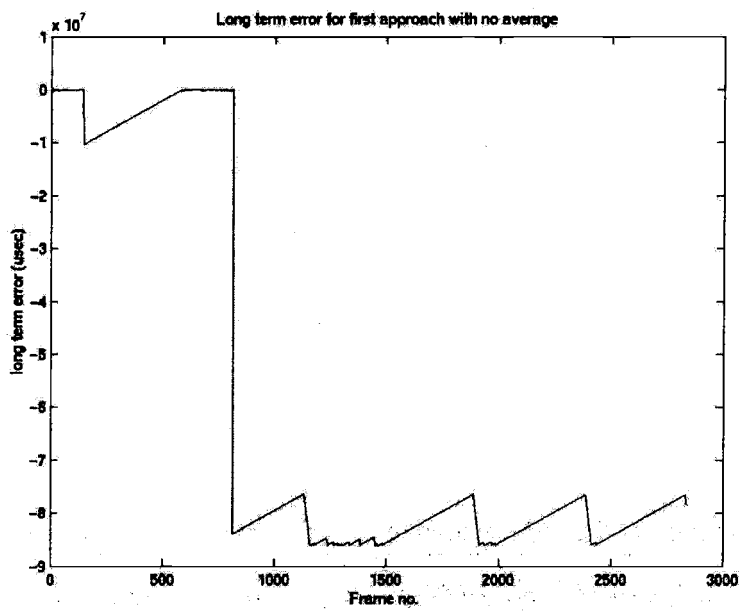
5.2.2 First controller design approach

In traditional controller design, we apply control to correct both positive and negative errors. This means that when we want to control a certain quantity to a zero value, we apply positive control effort when the error is negative and apply negative control effort when the error is positive. We cannot apply negative allocation in computer systems. Zero allocation is the closest value to the negative control effort used by a traditional controller.

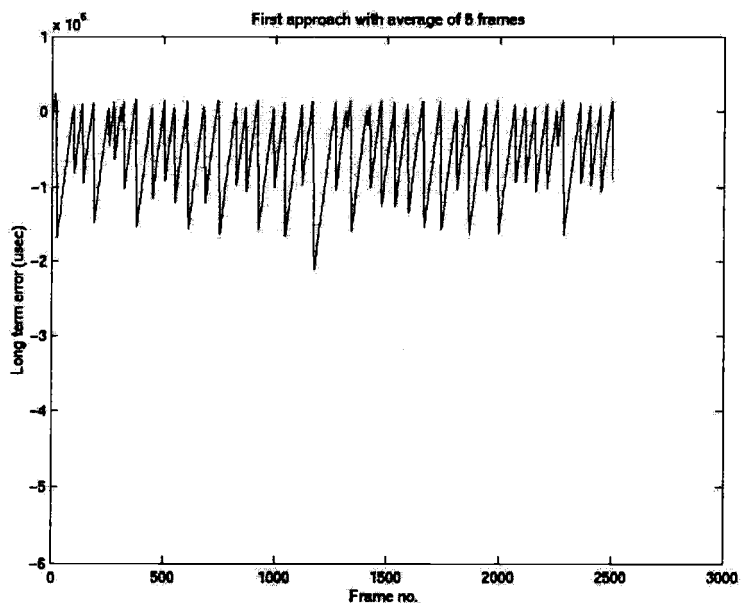
In our first approach, we tried our controller with the allocation set equal to the cycles calculated by the controller when the error was negative (i.e., when the video lagged real-rate) and zero allocation when the error was positive (i.e., when the video was ahead). We implemented this controller first with no averaging of k and then with averaging. Plots of the results of this approach are shown in Figure 5.4.[1] We used the optimal controller design from Section 4.2.

[1] When the video stops for some time, the application does not write properly to the kernel logs. In these plots, many data points in between are missing. The plots give a rough idea about the large error variations experienced.

Figure 5.4 Plots of long-term error (μsec) for first approach with traditional controller

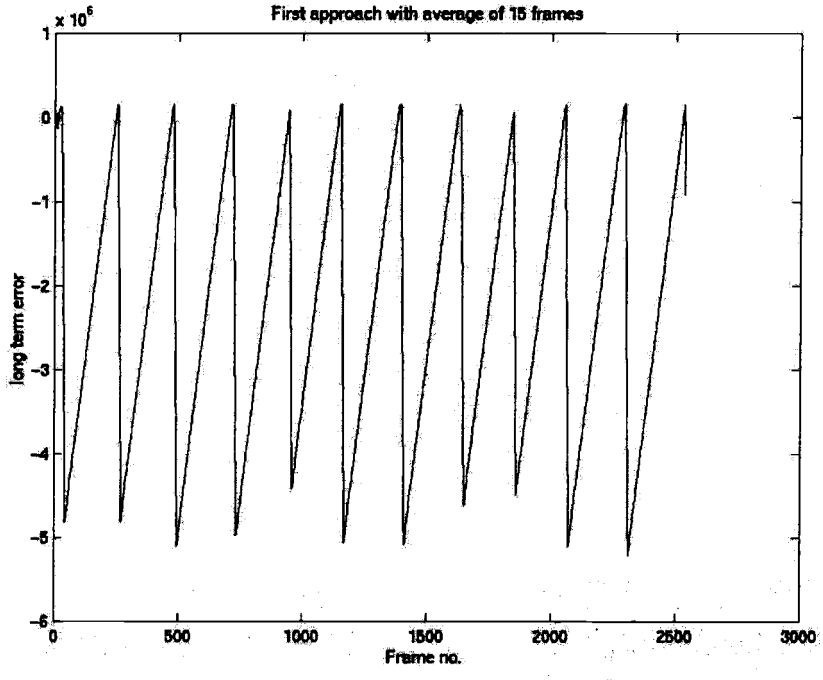


(i) No averaging



(ii) Average of 5 frames

(iii) Average of 10 frames



(iv) Average of 15 frames

The plots show that this approach does not work well. There are large variations in the long-term error as the controller switches between using zero control and using positive control. The video becomes jittery and, in some cases, stops for a long time. The reason for this poor performance is explained below.

Problems with this approach:

When the error is positive, zero or near zero, the allocation for video decoding becomes very small or zero. During the next sampling period, after running for a short while, or not at all, the decoder thread gives away the CPU to another application because the error appears to be properly controlled to zero. However, the video still requires some CPU to keep making progress. Since our kernel is non-preemptive, the other application may run for a long time before relinquishing control of the CPU. This causes our application to stop running for more than one sampling period. No

frame is processed during this period, and the error keeps increasing. This causes a large negative error. The allocation needs to be large the next period, and an oscillation occurs in both the allocation and error. In the no-averaging case, sometimes the application stops for a very long time. The application is not able to recover from this stage. The video stops for a long while and then restarts again.

Averaging improves the response but does not fix this problem when it is used. Recall that the controller gains are updated only at the end of each averaging window. The oscillation is more pronounced while using this controller setup with averaging since the controller will not start adjusting for the increased error until its next averaging window if the video is running too far ahead. The video does not stop for a long while in this case but becomes jittery.

Some of the choices made in the controller design:

When averaging over a window, we average the *k* value for the length (in frames) of a window, use the corresponding optimal gain for next entire window length, then get the average of the new window and update the control gain again, and so on. This is illustrated in Figure 5.5. This reduces the overhead since we do not have to calculate the optimal control gain every sampling period. Each sampling period, we just apply the control gain to the new measured errors to compute the new allocation.

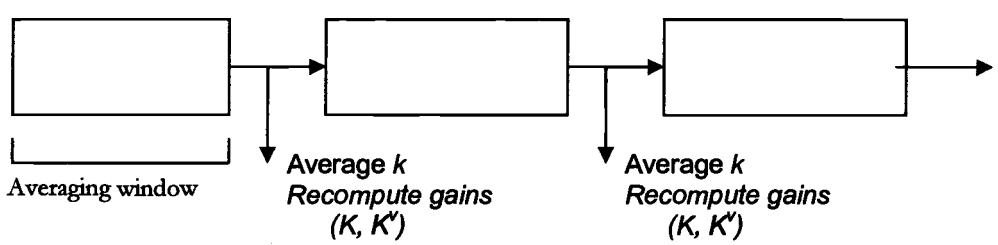


Figure 5.5 Averaging window

5.2.3 Second approach: fixed switching controller

We modify the controller so that it assigns some CPU even when the error is positive. From earlier experiments on MPEG video, we have a fairly good idea of the processing time requirements of various frames. We can use this information to assign the CPU while the error remains positive. We tested our controller with a fixed CPU allocation when the error was positive, zero or near zero. Figure 5.6 shows the flowchart for this approach.

There are two parts to this controller. One is just a fixed allocation, and the other is the optimal control part. The controller switches between these two, depending on the error. The optimal control is applied when the error is significantly negative, and the fixed control is applied when the error is positive or close to zero. We call this controller a *Fixed Switching Controller* since it switches between a fixed allocation part and the optimal control part.

Plots showing the performance of this approach are shown in Figure 5.7. In this case, averaging k does not improve the response much. This may be due to the fact that fixed allocation part does not respond to changes in requirements within the video. The switching is apparent in the performance plot when there is no averaging. The optimal controller controls the error more closely than the fixed allocation controller does. This is expected since the fixed allocation part does not change the allocation when the error changes as long as the error remains positive. Thus, the error will have large variations as long as the fixed allocation is in charge. Large variations in the error on the plots indicate that fixed allocation was dominant. Relatively small variations indicate that optimal control was also active a significant percentage of the time. This is indeed the case. In the first plot, in the regions where the error has smaller variations, the optimal control and the fixed allocation were called almost equally. In the regions where the error has larger variations, the fixed allocation was selected more often (more than 75%) than optimal control part.

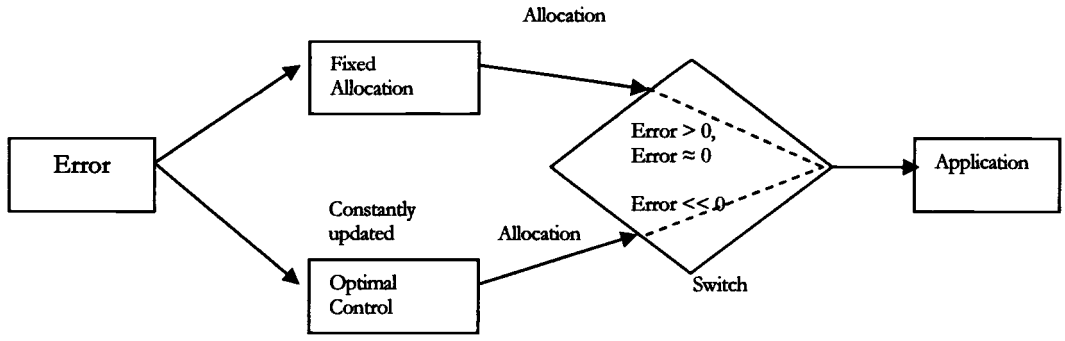


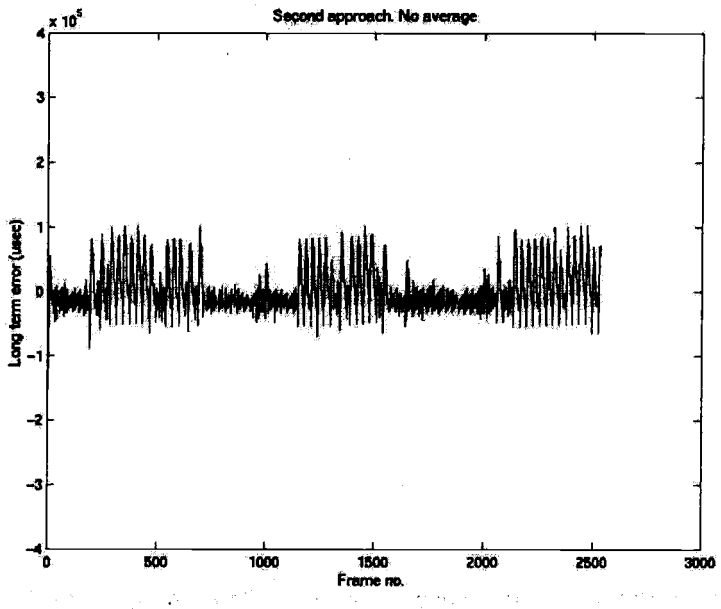
Figure 5.6 Fixed switching control

Problems with this approach

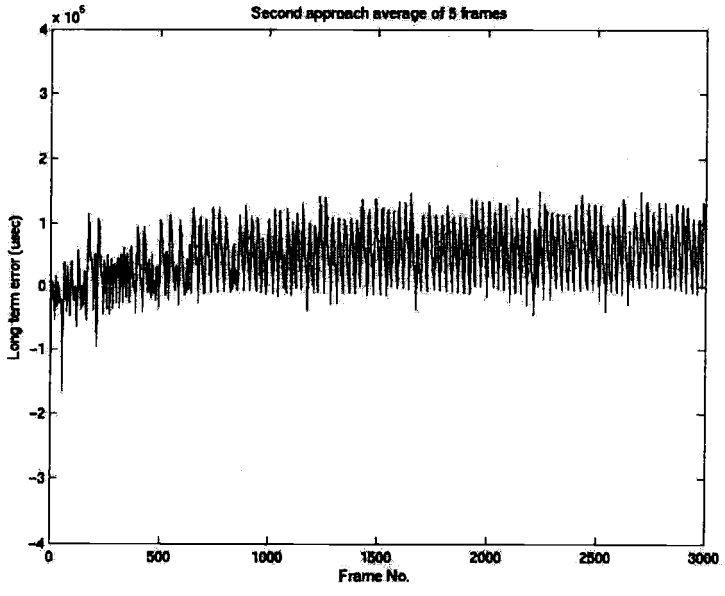
This approach has the limitation that we must select the value of the fixed allocation beforehand. For this, we need to have data on the processing time requirements of the particular video. This *a priori* information may not always be available. We can use information collected from other videos, but there is no guarantee that it will work for the current video. Our MPEG measurements show that this is true. This approach is thus not well suited for real-rate flows.

Another problem, apparent from plots and discussed earlier, is the inability of the fixed control part to keep the variations of the long-term error low. These variations can be smoothed by using averaging for the fixed allocation part the same way that averaging was used to estimate *1/k*. This can be done online when the error is positive by estimating the average required fixed allocation over some window.

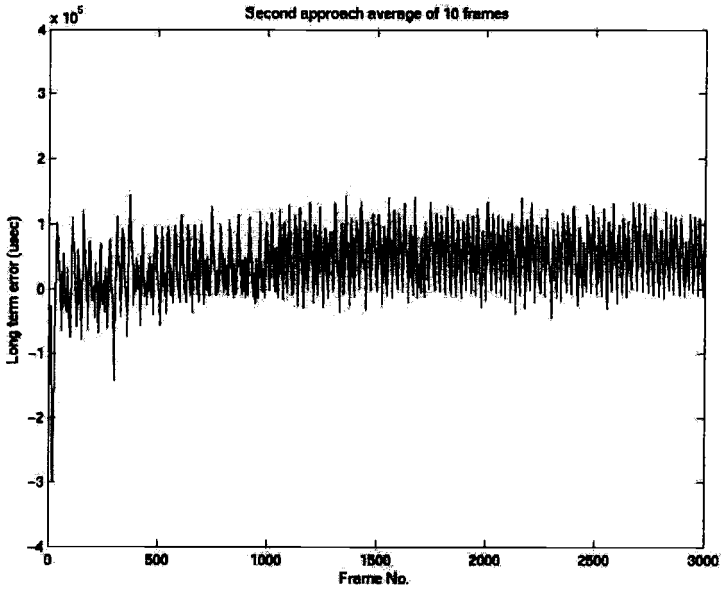
Figure 5.7 Plots of long term error (μsec) for second approach with fixed switching control



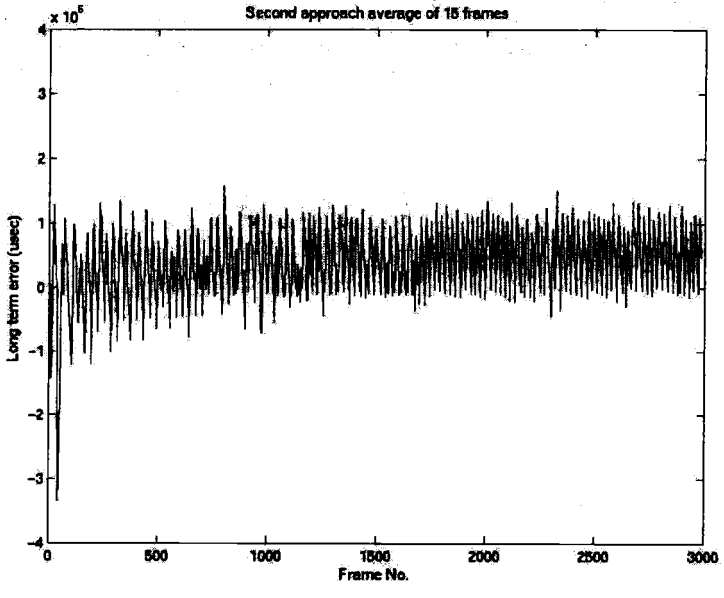
(i) No averaging (around 55 % fixed allocation when there is less variation in the plot, and 75 % of fixed allocation when there is more variation in the plot)



(ii) Average of 5 frames (around 75% fixed allocation)



(iii) Average of 10 frames (around 84% fixed allocation)



(iv) Average of 15 frames (around 86% fixed allocation)

5.2.4 Third approach: estimating switching controller

As discussed in the last section, we estimate the allocation needed, when the error is positive, by taking the average of the processing requirements of the most recently decoded video frames. For this, we tried using the same length averaging window as considered earlier for averaging *1/k*. This modified controller is shown in Figure 5.8.

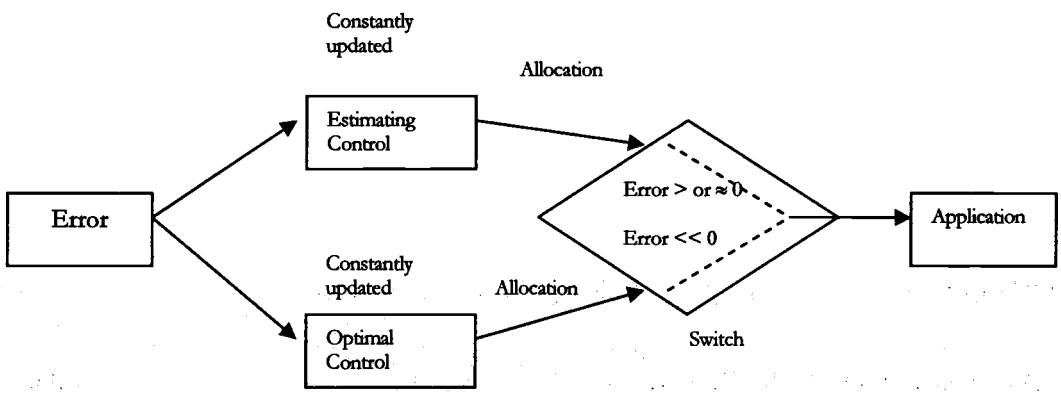
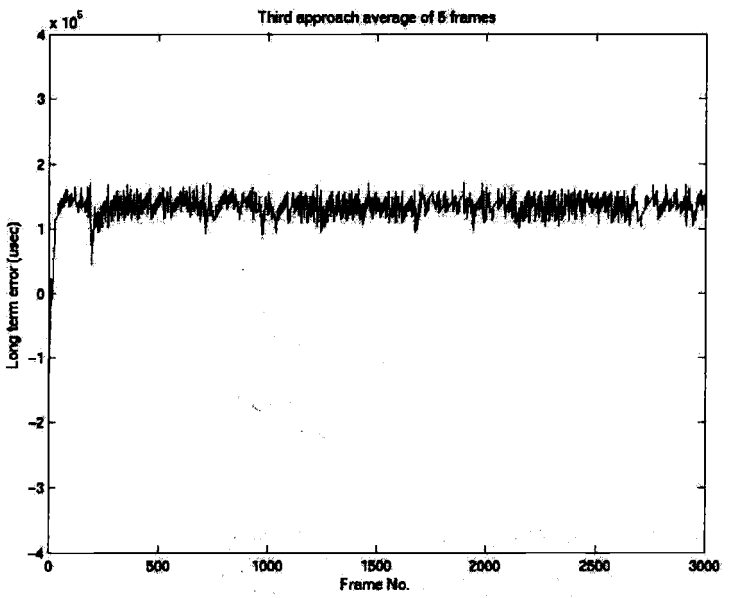


Figure 5.8 Estimating switching controller

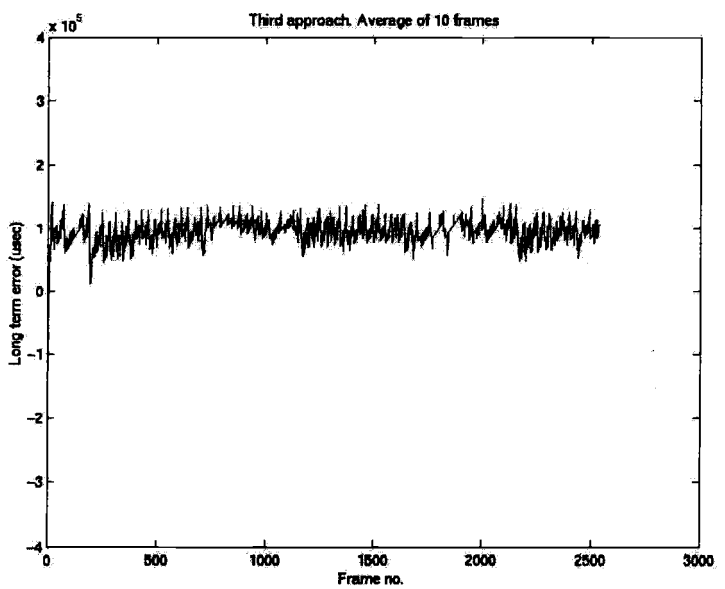
In this switching controller, the optimal controller takes over when the error is significantly negative and assigns the CPU to drive the error towards zero. When the error is close to zero or positive, the estimating control takes over to keep the error close to zero. We call this controller an *Estimating Switching Controller.*

The performance results for the estimating switching controller with different averaging window lengths for the estimating controller part are shown in Figure 5.9.

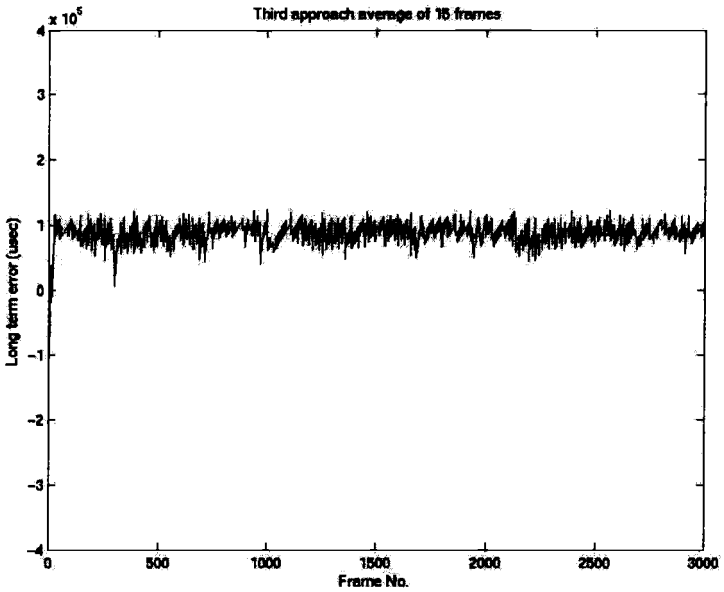
Figure 5.9 Long-term error (μsec.) for estimating switching controller for bike.mpg video with different averaging windows



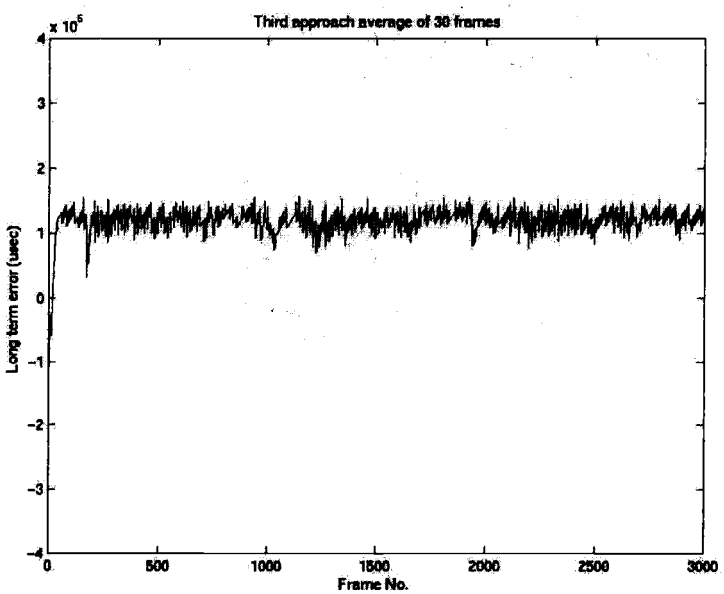
(i) Average of 5 frames



(ii) Average of 10 frames



(iii) Average of 15 frames



(v) Average of 30 frames

As can be seen from the various plots in Figure 5.9, the variations in the long-term error gradually smooth out as we continue increasing the averaging window. The optimal controller tries to compensate for the negative error. During the next few sampling periods, it assigns more CPU and the error starts to decrease and eventually becomes positive. At this time the estimating controller takes over. When there is less averaging, for example an averaging window length of 5 frames, the estimating controller tries to compensate for short-term variations in the error. This causes some variations in the error around zero. As we start increasing the averaging window, the magnitude of these variations gradually decreases as there becomes less overcompensation of the estimating control. When we reach the window length of around 15 frames, these variations become significantly less. At this stage, only the effects of the variations in k are left, and the variations due to the overcompensation are almost nullified. This result is consistent with our earlier hypothesis that the length of GOP should be a good choice for the length of the averaging window.

From Figure 5.9, notice that we also have some variation of the error at the start of the video. These variations increase as we increase the averaging window length. This is due to the fact that we did not average decoding time until the first window length was reached. We used a fixed allocation during this period, which accounts for the large variations in the error.

We hypothesized that the length of the averaging window had an optimal value. If the length of the averaging window increased beyond this threshold, we expected the variance in the long-term error to increase. The reason behind this is that, if we try to average beyond the GOP boundary, there will again be a different mix of frame types in each window of frames. For example, if averaging length is increased one unit from GOP boundary, next I frame will be included in the average if the window starts at the previous I frame, but only one I frame will be included in the average if the window starts after the previous I frame. Since the I frames are significantly different in size than the other frames, and since we found that the decoding time was related linearly to frame size, we expected that this would increase the variation of the averages significantly. We expected that the length of averaging window should be optimal around multiples of the GOP length. Figure 5.10 shows the variance of the long-term error for different lengths of the averaging window.

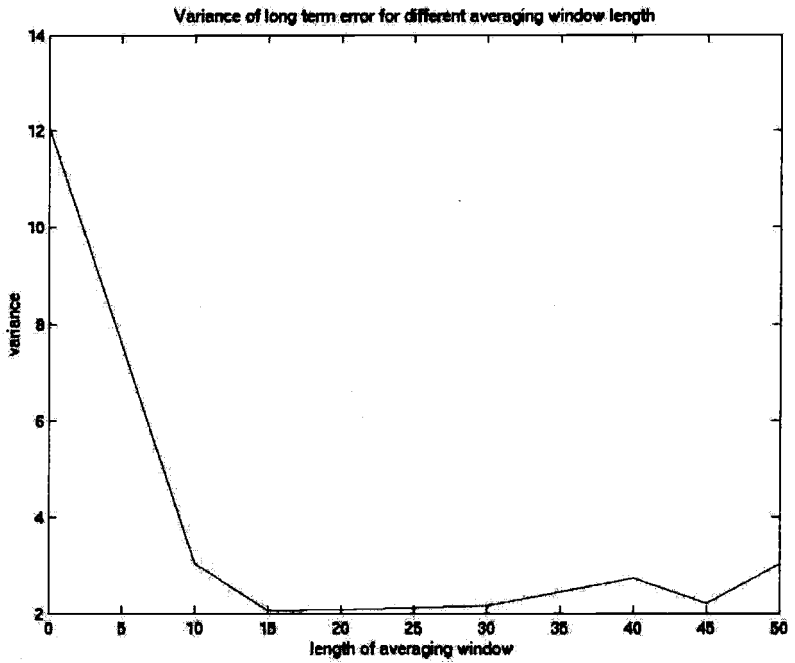


Figure 5.10 Variance in the long-term error for different lengths of the averaging window (Note: Only the data after the initial variations is considered)

As expected, the optimal averaging window length is 15 frames, where the variance in the long-term error is the minimum. As we increase the window further, the variance increases slightly. The variance again decreases when we reach any window length that is a multiple of 15 (the GOP length).

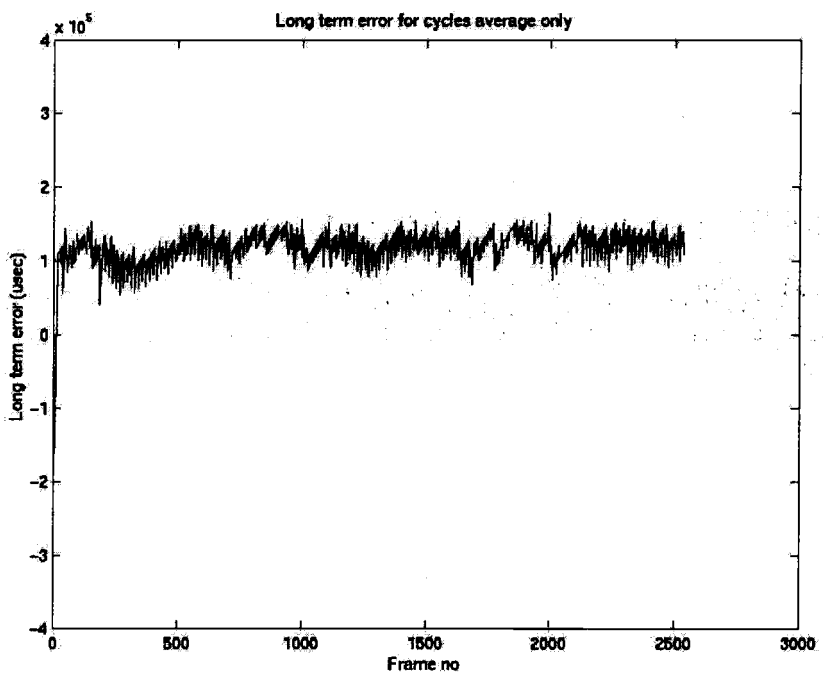
Figure 5.9 shows that we have close control of the long-term error with this controller. There is no accumulation of error and the variations in instantaneous error are quite small. Variations are reduced, from around 30 msec with no averaging, to less than 15 msec with a window length of 15. Next, we resolve the issue of the large initial variations. This is addressed later, in the fifth approach.

One interesting question that arises from this implementation is how well the estimating control will perform alone, in the absence of the optimal part. We discuss this approach in the next section.

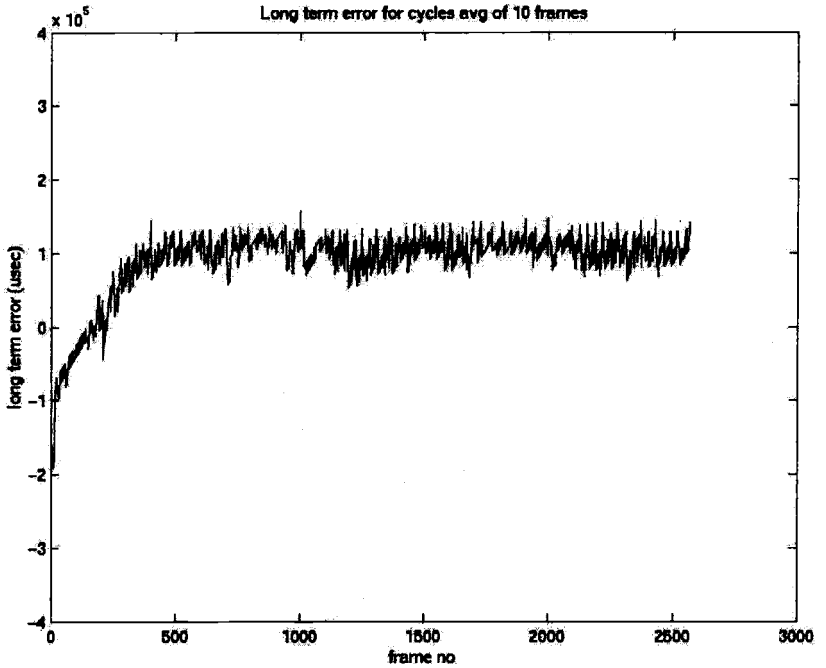
5.2.5 Fourth approach: estimating controller only

In this approach, we use only the averaging of the previous cycles. We do not use optimal control at all. We wanted to see how well this controller would perform without the optimal control. Would the controller perform satisfactorily if we do not use optimal control? The performance results are shown in Figure 5.11.

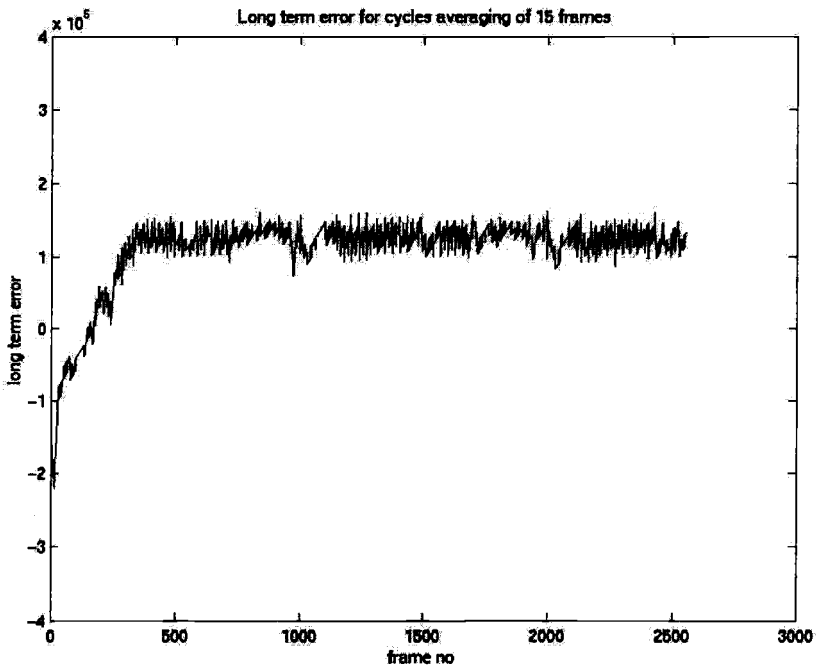
Figure 5.11 Long term average (μsec) for only estimating controller, with no optimal control.



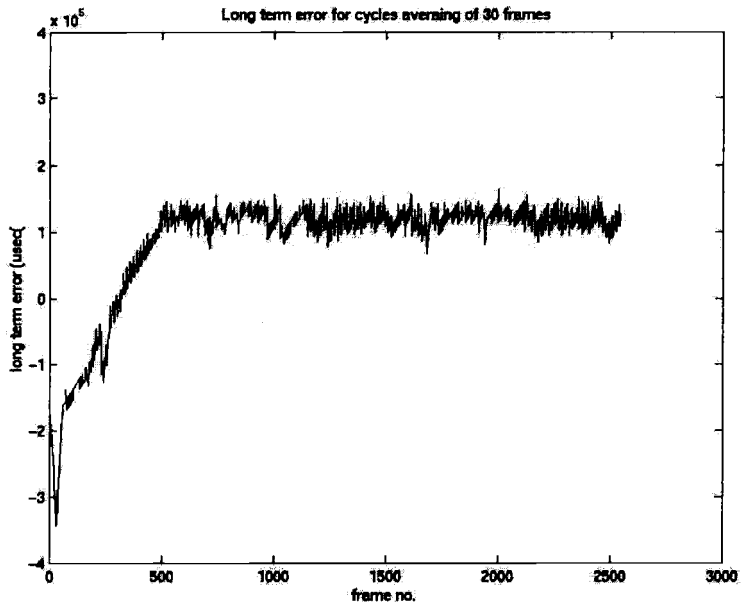
(i) Average of 5 frames



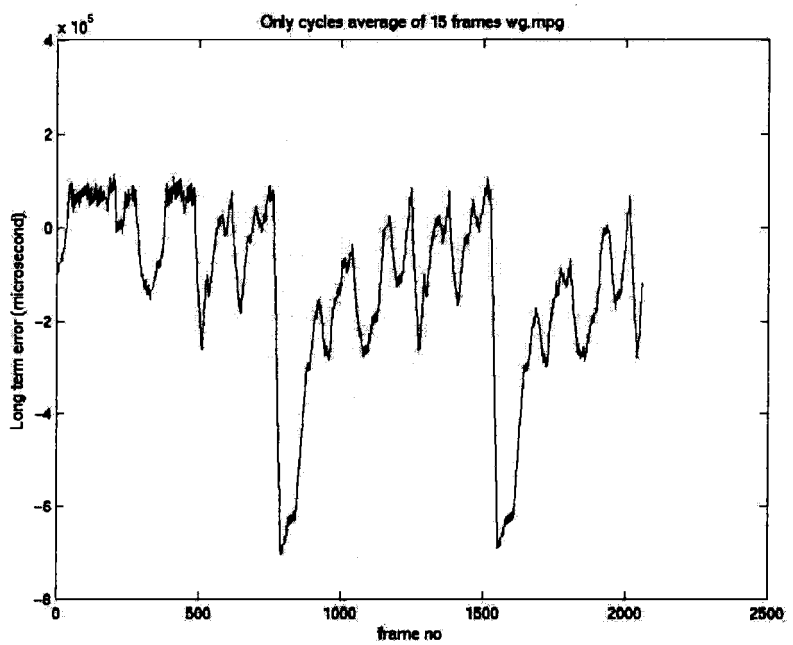
(ii) Average of 10 frames



(iii) Average of 15 frames



(iv) Average of 30 Frames



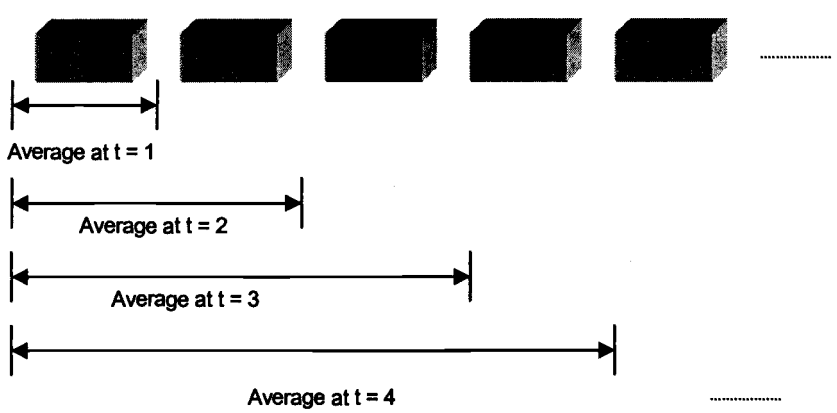
(v) Average of 15 frames for wg.mpg video

As can be seen from the plots, the controller works fine when we have an averaging window of only 5 frames. But as we increase the averaging window further, the settling time for the controller becomes large. This is because as we do not have an optimal (feedback) control. We just have the estimating (feedforward) controller. The controller can not adjust to large negative errors by increasing the gain quickly. Hence, the error decreases slowly and settles to its final value after some time. This results in a long response time of the controller to large instantaneous errors. This effect is more pronounced for wg video as shown Figure 5.11(v). This video has rapid scene changes. CPU requirements of this video have large jumps. Controller has to quickly adjust to these changes. For this video we do not get satisfactory performance from this controller.

5.2.6 Final approach (estimating switching controller with averaging at start)

In our third approach, we did not use any averaging at the start. We used averaging only after the first averaging window length was reached. This resulted in large variations in the error at the beginning of the video, as discussed in Section 5.2.4. This is apparent in the plots for the third approach in Figure 5.8. In order to reduce these initial variations, we now use progressive averaging until the first window length is reached. We keep updating the average to include the current frame number, as data becomes available. We expect that this progressive averaging will reduce the large variations at the start. This averaging at the start is illustrated in Figure 5.12.

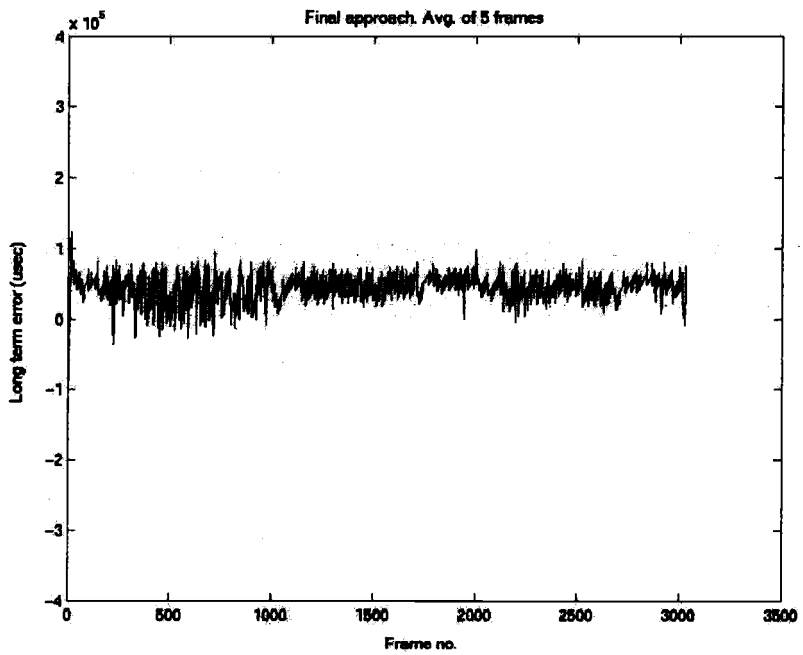
Figure 5.12 Progressive averaging



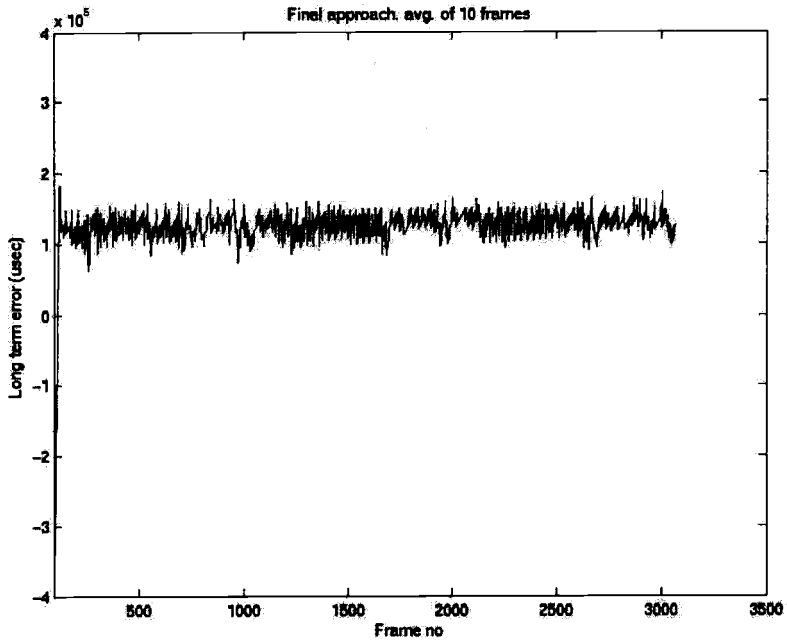
We update the estimating controller every sampling period at the start of the video unlike the usual sliding window averaging that we use later in the video, where we update the estimating controller only every multiple of the window length. This will increase the overhead slightly during startup.

The flowchart of this controller is shown in Figure 5.15. Results of this approach are shown in Figure 5.13.

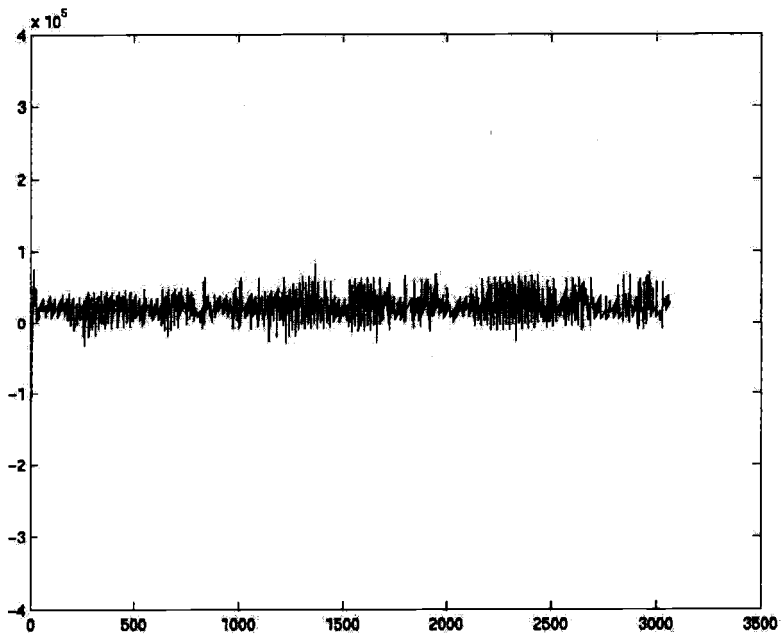
Figure 5.13 Third approach with progressive averaging added at start



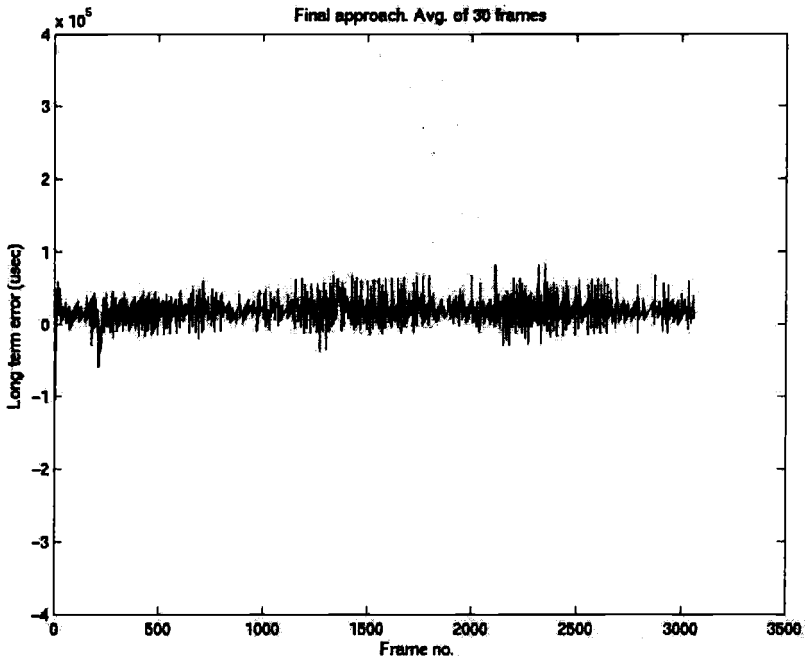
(i) Average of 5 frames



(ii) Average of 10 frames



(iii) Average of 15 frames



(iv) Average of 30 frames

As we can see from Figure 5.13, we have less variation of the error at the beginning of the video with this approach. We still have some variations, but they diminish very quickly. This is apparent in the plots for the averaging window of 30 and 50. In this approach, the initial variations diminish by 10 frames, while in the third approach these variations take more than 50 frames to diminish.

Result of running this controller on wg video is shown in Figure 5.14. As mentioned earlier wg video has very rapid scene changes. If our controller works for this video, we can be assured that it will work for any video. This is confirmed in Figure 5.14. Variations in error are quite small.

Figure 5.15 shows the variances of the different approaches for a window length of 30 frames.

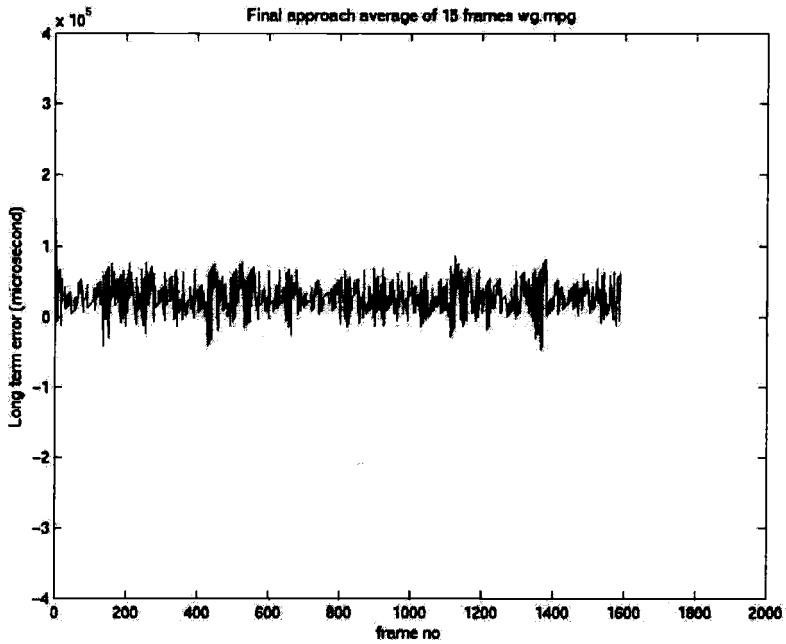


Figure 5.14 Long term error for wg.mpg video

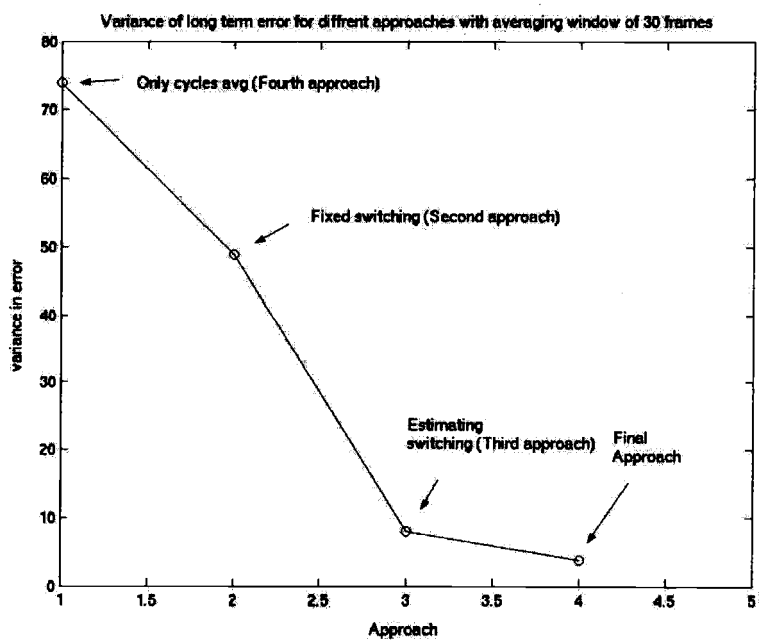


Figure 5.15 Variance in long term error for different approaches with window length of 30 frames.

As we can see from the figure, the variance in long-term errors decreases from around 75 when using the estimating only approach (approach four) to around 4 when using the final approach. The variance for the estimating switching controller decreases from 7.96 to 3.75 when we add averaging at startup to the third approach.

5.2.7 Buffering requirements

The performance of this real-rate controller will have a significant impact on the buffering requirements at both ends of the decoder application. At the output side, we need a -buffer length that will absorb the variations in the frame-decoding rate. These rate variations can be seen in the plots of the long-term error. We see from the plots of the error in our final approach that the variations in error are less than 100 milliseconds. Since frames are 33 milliseconds apart, we will require the buffering of around 3 to 4 frames.

5.3 Conclusion and Accomplishments

This is our first attempt to apply dynamic optimization to this computer system resource allocation problem. Initial results are encouraging and more dedicated research in this area can go a long way in helping computer system designers. We can see from the results that despite quantization in our systems we have been able to keep variations in error quite small. This is very encouraging result. Normally complex control schemes have large computation requirements. This can result in large overhead in the system. In our system, however, we were able keep overhead quite small. This was due to simple mathematical model and doing most of the expensive calculations offline. In fact, final implementation of the controller requires just a few multiplications and additions.

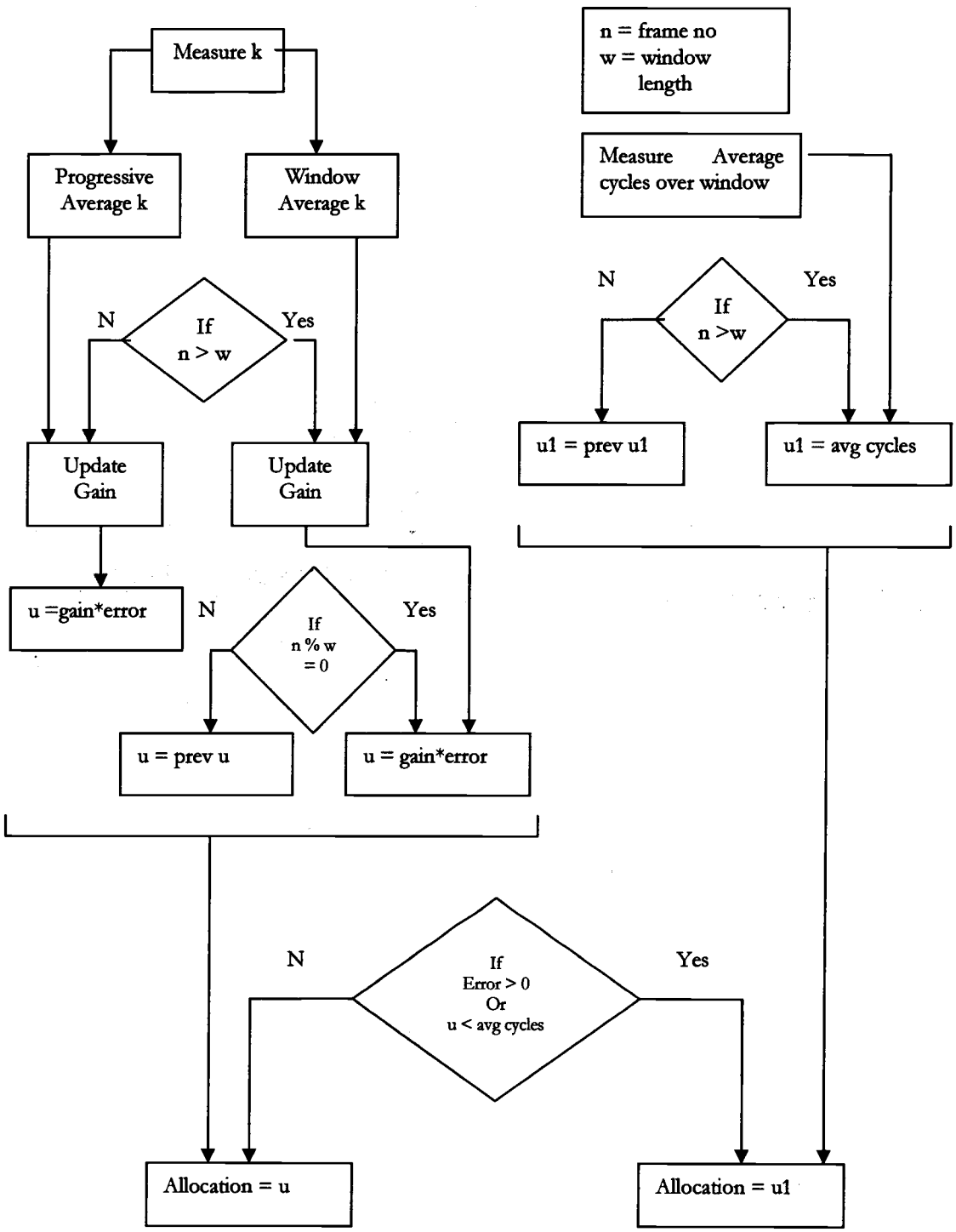
The related work of Luca [1] uses *a priori* information on the type of video frames to be decoded to allocate CPU for each frame in a real-time O/S. Our goal is to determine control strategies without use of a priori information.

Main accomplishments of this thesis are listed below.

Accomplishments

- Identified various issues like quantization that need to be addressed for multimedia applications.
- Designed a feedback controller for allocating CPU soft real-time multimedia applications like MPEG videos.
- Implemented this controller on TSL.
- Tested this controller on various MPEG videos.
- Analyzed statistically MPEG videos for various properties like frame size, decoding time etc.

Figure 5.15 Flowchart of controller for the final approach



(This module is run whenever a new frame has been decoded)

Chapter 6

Related Work

In this chapter, we discuss the research work done elsewhere that is related to our research. First, we discuss CPU scheduling in real-time operating systems. Then we discuss real-time scheduling in general-purpose operating systems, and last we discuss feedback-based scheduling.

6.1 Real-time Scheduling Algorithms

Extensive research has been devoted to the problem of scheduling in real-time operating systems [17,18,19]. There are mainly two classes of algorithms used for scheduling in real-time operating systems. These algorithms are priority-based scheduling and proportion-period scheduling. The main consideration made in the design of these algorithms was to ensure that the sum of all requested resources be less than the total available resource.

These algorithms make several simplifying assumptions about an application's resource requirements. They assume that threads have periodic deadlines and require constant execution time in each period. They also assume that threads give up CPU voluntarily and the system is fully preemptible. Lastly, they assume that threads are independent of each other.

6.1.1 Priority-based Scheduling

In priority-based scheduling, the scheduler assigns real-time priority to each thread based on its execution time requirements and its deadline [17]. The scheduler then selects the thread with the highest priority for execution. The scheduler can assign these priorities statically or dynamically. An example of a static priority scheduling is *rate-monotonic* (RM) scheduling, and an example of dynamic priority scheduling is *earliest deadline first* (EDF). These are the most widely used priority-based real-time scheduling algorithms in real-time operating systems.

In rate-monotonic scheduling, the scheduler assigns a static priority to each thread in such a way that threads with smaller periods get higher priority. In this scheduling algorithm, the thread with the smallest period gets the highest priority. One problem with this approach is that it leads to starvation of threads with large scheduling periods.

In the earliest-deadline-first algorithm, the scheduler assigns a priority based on the deadline of each thread. The thread with the closest deadline gets the highest priority and therefore is executed first. The processor can be fully utilized with EDF scheduling [17]. EDF algorithms, however, have more overhead than the RM algorithm.

6.1.2 Proportion-period Scheduling

In priority-based scheduling, the highest priority thread gets control of the CPU. This approach can run into trouble if misbehaving threads with high priority do not yield the CPU. These misbehaving threads can starve other lower-priority threads. To alleviate this problem, real-time operating systems must provide *temporal protection* to threads so that misbehaving threads with large execution times do not affect other threads. Proportion-period schedulers were designed to solve this problem.

In a basic proportion-period scheduler, each thread is allocated a fixed proportion of the CPU every scheduling period. This proportion is determined based on the needs of each thread at each scheduling period, such that the total allocated CPU proportion is less than the total available CPU resource. The period of the scheduler can also be varied based on an application's delay requirements. The period defines a repeating deadline of an application. If the scheduler cannot allocate enough CPU required by thread before the deadline (the end of the period), then the thread will miss the deadline.

6.2 Real-time Schedulers in General-purpose Operating Systems

In real-time operating systems, applications express their timing constraints to the operating system, and the operating system provides execution control for scheduling. The scheduling analysis of real-time operating systems ignores issues inherent in general-purpose operating systems, such as kernel non-preemptibility and interrupt overhead. Recently, several real-time algorithms have been implemented on Linux and other general-purpose operating systems. For example, RED Linux provides a generic scheduling framework for implementing different real-time scheduling algorithms [31]. A different approach for providing real-time performance is used by other systems, such as RTLinux [31]. RTLinux decreases the unpredictability by running Linux as a background process over a small real-time executive. Real-time threads are not Linux processes in this case but run on a real-time executive, and Linux kernel runs as a non real-time thread. This solution provides good real-time performance, but Linux processes are still non real-time. This solution will not provide good performance to user-space real-time processes.

Most general-purpose operating systems, such as Linux, Solaris and NT, provide real-time priorities. These real-time priorities are still priorities in the conventional sense, but are higher than conventional application thread priorities. These priorities are not derived from the resource requirements of real-time threads and hence they do not provide good control. Several proportional-share scheduling mechanisms have been implemented in various operating systems, such as Linux, FreeBSD, Solaris and Windows [20, 21, 22, 23, 24, 25]. These approaches require external inputs regarding the resource requirements of the threads and focus on the best way to satisfy these requirements. None of these approaches infer requirements of threads dynamically from measurements.

6.3 Feedback-based Scheduling

Feedback is widely used in operating-system schedulers to build adaptive operating systems. The Unix operating system uses multi-level feedback queue scheduling [27] to schedule processes. The operating system monitors each thread to see whether the thread uses its entire allocated time slice, or whether it stalls waiting for an input-output (I/O) resource to become available, and adjusts the thread's priority accordingly. I/O-bound processes are given a higher priority over CPU-intensive processes. This ensures that interactive processes do not suffer because of CPU-intensive processes.

The feedback scheduling in TSL was mainly influenced by Massalin and Pu's work. Massalin and Pu were the first to propose the use of feedback control for fine-grained resource management in operating systems [28]. They used fine-grained scheduling for interdependent jobs such as threads in a pipeline. An application is divided into a number of threads with queues separating each thread. Application data is passed along the pipeline and some processing work is done at each pipeline stage. The queue lengths at the input and output buffer of each stage were used as a measure of application's progress at that stage. The main problem with this approach was that the behavior at a pipeline stage was dependent on the behavior at other stages. A poorly-designed controller at one stage might fill up a queue affecting the behavior of a different thread. Goel, et al., modified this approach and used time-stamps on data packets as a measure of an application's progress rather than using queue fill levels [4]. This approach allows a controller at each stage to determine the progress of application at that stage independently of other threads in the pipeline.

Goel, et al., first used a fixed-gain PI controller as the control law. This approach caused problems for applications with large variations in their processing-time requirements. Later, they proposed a PI controller with two adaptive parameters, which could be varied based on the different application needs. These parameters could be tuned on-line so that the controller would perform well for a wide range of scenarios. Our feedback scheduler uses a more sophisticated control algorithm that is less sensitive to measurement error to adapt to changes in the processing requirements of the frames.

In Goel's PhD dissertation [13], which was completed in parallel with our research, he changed his control algorithm structure and the TSL scheduling period. However, his design approach remained an exhaustive search in the parameter space.

Abeni, et al., used a feedback controller to monitor and adapt the reservation granted to real-time threads in a real-time operating system [1]. A feedback scheme is used to estimate the resource requirements of real-time applications. This estimated requirement is then used to reserve CPU for the real-time threads using proportion-period scheduling. Their feedback controller is essentially a PI controller.

Bibliography

[1] Luca Abeni, Luigi Palopoli, Guiseppe Lipari and Jonathan Walpole, "Analysis of a Reservation-Based Feedback Scheduler," *Proceedings of the Real Time Systems Symposium (RTSS 2002),* Austin, Texas, December 2002.

[2] Charles Krasic and Jonathan Walpole, "Quality-Adaptive Media Streaming by Priority Drop," *23rd International Conference on Distributed Computing Systems, (ICDCS 2003),* Providence, Rhode Island, May 19-22, 2003.

[3] Buck Krasic and Jonathan Walpole, "Priority-Progress Streaming for Quality-Adaptive Multimedia," *Proceedings of the ACM Multimedia Doctoral Symposium,* Ottawa, Canada, October 2001.

[4] Ashvin Goel, Molly H. Shor, Jonathan Walpole, David C. Steere, and Calton Pu, "Using Feedback Control for a Network and CPU Resource Management Application", *Proceedings of the 2001 American Control Conference (ACC),* June 2001.

[5] Ashvin Goel, Luca Abeni, Jim Snow, Charles Krasic, Jonathan Walpole, "Supporting Time- Sensitive Applications on General-Purpose Operating Systems," *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI),* Boston, MA, December 2002.

[6] Quasar project software releases available at http://www.cse.ogi.edu/DISC/projects/quasar/releases/

[7] Luca Abeni, Ashvin Goel, Buck Krasic, Jim Snow, and Jonathan Walpole, "A Measurement- Based Analysis of the Real-Time Performance of the Linux Kernel", *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002),* San Jose, California, September 2002.

[8] Ashvin Goel and Jonathan Walpole, "Gscope: A Visualization Tool for Time-Sensitive Software", *Proceedings of the Freenix Track of the 2002 USENIX Annual Technical Conference,* Monterey, California, June 2002. Software release available at http://gscope.sourceforge.net/.

[9] Ashvin Goel, David Steere, Calton Pu, and Jonathan Walpole, "SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit", *Proceedings of the 2nd Usenix Windows NT Symposium*, September 1998.

[10] David Steere, Molly H Shor, Ashvin Goel, Jonathan Walpole and Calton Pu, "Control and Modeling Issues in Computer Operating Systems: Resource Management for Real-Rate Computer Applications", *Proceedings of 39th IEEE Conference on Decision and Control (CDC)*, December 2000.

[11] Thrasyvoulos Pappas, Alan j. Laub and Nils R. Sandell Jr. "On the Numerical Solution of the Discrete-Time Algebraic Riccati Equation", *IEEE Transactions on Automatic Control*, Vol. AC- 25, No.4, August 1980.

[12] A. Laub, "A Schur Method for Solving Algebraic Riccati Equations", *IEEE Transactions on Automatic Control*, AC-24 1979.

[13] Ashvin Goel, "Operating Support for Low-Latency Streaming", Dissertation, October 2002.

[14] John Wiseman, "An Introduction to MPEG Video Compression", http://www.mpeg.org.

[15] MPEG-1 and MPEG-2 Digital Video Coding Standards, http://www.mpeg.org.

[16] MPEG video compression standard edited by Joan L. Mitchell, New York : Chapman & Hall, c1997.

[17] C. L. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, Jan 1973.

[18] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[19] Lui Sha, Raghunathan Rajkumar, and John Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1184, September 1990.

[20] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *First Symposium on Operating System Design and Implementation*, pages 1–12, November 1994.

[21] Ian Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

[22] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical CPU scheduler for multimedia operating system. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 107–121, Berkeley, CA, USA, October 1996. USENIX.

[23] M. B. Jones, D. Rosu, and M.-C. Rosu. Cpu reservations and time constraints: efficient, predictable scheduling of independent activities. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, pages 198–211, Oct 1997.

[24] Jason Nieh and Monica Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Symposium on Operating Systems Principles*, October 1997.

[25] David K. Y. Yau and Siman S. Lam. Adaptive rate controlled scheduling for multimedia applications. *IEEE/ACM Transactions on Networking*, August 1997.

[26] I. Stoica, H. Abdel-Wahab, , and K. Jeffay. On the duality between resource reservation and proportional share resource allocation. In *Multimedia Computing and Networking*, volume 3020, San Jose, CA, feb 1997.

[27] F. J. Corbato, M. Merwin-Daggett, and R. C. Daley. An experimental time-sharing system. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 335–344, 1962.

[28] Henry Massalin and Calton Pu. Fine-grain adaptive scheduling using feedback. *Computing Systems*, 3(1):139–173, Winter 1990. Special Issue on selected papers from the Workshop on Experiences in Building Distributed Systems, October 1989.

[29] Quasar Software Package. http://www.cse.ogi.edu/sysl/projects/quasar/releases/.

[30] A. C. Bavier, A. B. Montz, and L. L. Peterson. Predicting MPEG execution times. *Proceedings of SIGMETRICS '98/PERFORMANCE '98* (June 1998).