Managing the Effect of Delay Jitter on the Display of Live Continuous Media

Donald L. Stone

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1995

Approved by:

______________________________	Advisor
______________________________	Reader
______________________________	Reader

© 1995

Donald L. Stone

All Rights Reserved

DONALD L. STONE. Managing the Effect of Delay Jitter on the Display of Live Continuous Media (under the direction of Kevin Jeffay).

ABSTRACT

This dissertation addresses the problem of displaying live continuous media (*e.g.*, digital audio and video) with low latency in the presence of delay jitter, where delay jitter is defined as variation in processing and transmission delay. Display in the presence of delay jitter requires a tradeoff between two goals: displaying frames with low latency and displaying every frame. Applications must choose a display latency that balances these goals.

The driving problem for my work is workstation-based videoconferencing using conventional data networks. I propose a two-part approach. First, delay jitter at the source and destination should be controlled, leaving network transmission as the only uncontrolled source. Second, the remaining delay jitter should be accommodated by dynamically adjusting display latency in response to observed delay jitter. My thesis is that this approach is sufficient to support the low-latency display of continuous media transmitted over building-sized networks.

Delay jitter at the source and destination is controlled by implementing the application as a real-time system. The key problem addressed is that of showing that frames are processed with bounded delay. The analysis framework required to demonstrate this property includes a new formal model of real-time systems and a set of techniques for representing continuous media applications in the model.

The remaining delay jitter is accommodated using a new policy called queue monitoring that manages the queue of frames waiting to be displayed. This policy adapts to delay jitter by increasing display latency in response to long delays and by decreasing display

latency when the length of the display queue remains stable over a long interval. The policy is evaluated with an empirical study in which the application was executed in a variety of network environments. The study shows that queue monitoring performs better than a policy that statically chooses a display latency or an adaptive policy that simply increases display latency to accommodate the longest observed delay. Overall, the study shows that my approach results in good quality display of continuous media transmitted over building-sized networks that do not support communication with bounded delay jitter.

ACKNOWLEDGMENTS

First, I must thank my advisor, Kevin Jeffay. His guidance, advice, and patience made the most significant contribution to my success in graduate school. In addition, his friendship made the experience particularly rewarding.

Thanks as well to F. Don Smith whose help was extremely valuable. In addition to serving on my committee, he helped to obtain funding and equipment, participated in the initial design and implementation of the system, and conducted the experiments performed on the IBM network.

Thanks to all the students, staff, and faculty of the Computer Science Department. In particular, thanks to the other members of my committee, Don Stanat, Jim Anderson, and Jan Prins, and to the other members of the DIRT project, past and present. One of my greatest rewards in graduate school has been the set of wonderful personal and professional relationships I have developed over the years.

Thanks to the IBM and Intel Corporations for their generous support in fellowships, money, and equipment. Thanks also to the alumni of the Computer Science Department whose generous contributions made the Alumni Fellowship possible.

Thanks to Dean Smith and the 1990-1995 Tar Heel basketball teams.

Thanks particularly to my wife Claire whose love and support have been more important to me than she can know. Of the many wonderful things I did during my time in graduate school, meeting and marrying her was the most wonderful.

Most of all, thanks to my father. Throughout my life, he has taught and inspired me, not least by imparting to me his love for Computer Science. Thus, it is altogether fitting that my dissertation work, like all the important things in my life, be dedicated to him.

TABLE OF CONTENTS

Page

Chapter I Introduction 1
1.1 Continuous Media 1
1.2 Delay Jitter 3
1.3 Research Approach and Contributions 5
1.4 Related Work 7
1.5 Dissertation Overview 13

Chapter II System Description 15
2.1 Introduction 15
2.2 Overview of the Application 16
2.3 Hardware Interrupts 17
2.4 YARTOS 18
2.5 Acquisition-Side Processing 25
2.6 Summary and Discussion 47

Chapter III Feasibility Analysis of YARTOS Task Systems 49
3.1 Introduction 49
3.2 System Model 51
3.3 The Effect of Interrupt Handlers 54
3.4 EDF/DDM Scheduling Discipline 57
3.5 Feasibility Conditions 60
3.6 Feasibility Test 65
3.7 Summary 70

Chapter IV Feasibility Analysis of the Acquisition-Side 71
4.1 Introduction 71
4.2 Modeling Hardware Interrupts 72
4.3 Reasoning about Request-Response Interrupts 74
4.4 Determining the Minimum Interarrival Time of Application Tasks 80
4.5 Feasibility of the Application 85
4.6 Summary 101

Chapter V Analysis of the Delay Bound 102
5.1 Introduction 102
5.2 Overview of Real-Time Logic.... 103
5.3 Basic Concepts.... 105
5.4 Correctness Conditions.... 109
5.5 Basic Axioms and Theorems.... 111
5.6 Task Descriptions.... 119
5.7 Bounded Delay Theorem.... 128
5.8 A Note on the Lower Bound 163
5.9 Discussion.... 164

Chapter VI Policies for Managing Delay Jitter....166
6.1 Introduction 166
6.2 Effect of Delay Jitter 167
6.3 Queue Monitoring 171
6.4 Summary.... 174

Chapter VII Evaluation of Delay Jitter Management Policies....175
7.1 Introduction 175
7.2 Description of the Study.... 176
7.3 Evaluating Delay Jitter Management Policies.... 181
7.4 Comparison of Queue Monitoring to the I- and E- Policies.... 185
7.5 Effect of the Threshold Parameter 190
7.6 Discussion and Summary.... 196

Chapter VIII Conclusions and Contributions 198
8.1 Thesis Summary 198
8.2 Conclusions.... 200
8.3 Contributions 200
8.4 Future Work.... 201

References 205

LIST OF FIGURES

Page

Figure 1-1: A Pipeline View of Continuous Media Processing..2
Figure 1-2: Hardware Environment..6
Figure 2-1: Table of Hardware Interrupts..18
Figure 2-2: Interrupt Handler Declarations..19
Figure 2-3: Application Task Declarations 19
Figure 2-4: YARTOS System Calls...21
Figure 2-5: Architecture of Example YARTOS Application 22
Figure 2-6: Example YARTOS Application 23
Figure 2-7: Audio and Video Buffers 27
Figure 2-8: Memory Management Calls 27
Figure 2-9: Audio and Video Operations..27
Figure 2-10: Operations on Queues...27
Figure 2-11: Network Transmission Declarations..28
Figure 2-12: Global Variable Declarations..28
Figure 2-13: High-Level Architecture 29
Figure 2-14: High Level View of the Video Process...31
Figure 2-15: Digitization Process..32
Figure 2-16: High Level View of the Audio Process...33
Figure 2-17: High Level View of the Transport Process..34
Figure 2-18: Fragment of the Video Process 36
Figure 2-19: Video Fragment Divided Into Tasks...36
Figure 2-20: Software Architecture of the Acquisition-Side 38
Figure 2-21: Pseudo Code for VBI Task...41
Figure 2-22: Pseudo Code for VBI1 Task...42
Figure 2-23: Pseudo Code for VBI0 Task...43
Figure 2-24: Pseudo Code for CC Task 43
Figure 2-25: Pseudo-Code for Audio Task..44
Figure 2-26: Pseudo-Code for Initiate_Send Task..45
Figure 2-27: Pseudo-Code for Transmit_Complete Task...46
Figure 4-1: Successive Executions of the Audio Task 73
Figure 4-2: Interval Between Odd/Even Pairs of Audio Tasks......................................74
Figure 4-3: Minimum Interarrival Time of Application Task Invocations.......................81

Figure 4-4: An Alternative View of the Acquisition-Side Architecture 86
Figure 4-5: Execution Costs 87
Figure 4-6: Summary of Interrupt Handlers 95
Figure 4-7: Summary of Application Tasks 95
Figure 4-8: Formal Definitions of the Interrupt Handlers 96
Figure 4-9: Formal Definitions of the Application Tasks 96
Figure 4-10: Formal Definitions of the Resources 97
Figure 4-11: Graph of Condition 1 98
Figure 4-12: Graph of Condition 2 for Vbi0 Task 99
Figure 4-13: Graph of Condition 2 for Initiate Send Task 99
Figure 4-14: Graph of Condition 2 for Packet Transfer Task 99
Figure 4-15: Graph of Condition 2 for Transmit Complete Task 100
Figure 4-16: Graph of Condition 2 for User Tick Task 100
Figure 4-17: Condition 2 for Keyboard Check Task 100
Figure 4-18: Graph of Condition 2 for Screen Output Task 101
Figure 5-1: Symbolic Constants 105
Figure 5-2: Relationships Among Symbolic Constants 106
Figure 5-3: Task Actions 106
Figure 5-4: Subtask Actions 106
Figure 5-5: Message Actions 107
Figure 5-6: Queuing Actions 107
Figure 5-7: Memory Management Actions 107
Figure 5-8: Video Frame Processing Actions 108
Figure 5-9: External Events 108
Figure 5-10: Correctness Conditions for a Video Frame 111
Figure 5-11: Actions Performed in Mutual Exclusion 116
Figure 5-12: At-Most-Once Actions 117
Figure 5-13: Main Theorem 129
Figure 5-14: Summary of Axioms and Theorems 131
Figure 6-1: I-Policy and E-Policy with Persistent Delay Jitter 168
Figure 6-2: I-Policy and E-Policy with Occasional Delay Jitter 170
Figure 6-3: Queue Monitoring Procedure 173
Figure 7-1: Basic Data (UNC Network) 178
Figure 7-2: Distribution of End-to-End Delay Jitter (UNC Network) 178
Figure 7-3: Basic Data (IBM-RTP Floor) 180

Figure 7-4: Distribution of End-to-End Delay Jitter (IBM-RTP Floor) 180
Figure 7-5: Basic Data (IBM-RTP Campus) 181
Figure 7-6: Distribution of End-to-End Delay Jitter (IBM-RTP Campus).................... 181
Figure 7-7: Comparison of I, E, and QM Policies (UNC Network)............................. 187
Figure 7-8: Comparison of I, E, and QM Policies (IBM-RTP Floor) 188
Figure 7-9: Comparison of I, E, and QM Policies (IBM-RTP Campus) 189
Figure 7-10: QM Policies with Varying Thresholds (UNC Network)........................... 191
Figure 7-11: QM Policies with Varying Thresholds (IBM-RTP Floor) 192
Figure 7-12: QM Policies with Varying Thresholds (IBM-RTP Campus) 192
Figure 7-13: QM Policies with Multiple Thresholds (UNC Network) 194
Figure 7-14: QM Policies with Multiple Thresholds (IBM-RTP Floor)........................ 195
Figure 7-15: QM Policies with Multiple Thresholds (IBM-RTP Campus).................... 196

LIST OF SYMBOLS

τ	A real-time task system.
I_i	An interrupt handler.
T_i	An application task.
R_i	A resource.
e_i	Cost of interrupt handler I_i.
a_i	Minimum interarrival time of interrupt handler I_i.
c_i	Cost of application task T_i.
U_i	Set of resources used by application task T_i.
d_i	Relative deadline of application task T_i.
p_i	Minimum interarrival time of application task T_i.
D_i	Minimum relative deadline among tasks that share a resource with T_i.
$f(l)$	Upper bound on time spent executing interrupt handlers in an interval of length l.
$\delta_i(l)$	Upper bound on number of invocations of T_i occurring in $[t, t+l]$
Ψ_τ	Achievable processor utilization of a task set τ.
B_τ	Max. value for which condition 1 of the feasibility conditions must be checked.
α_i	Lower bound on the response time of a request for interrupt I_i.
ω_i	Upper bound on the response time of a request for interrupt I_i.
MP_I	Set of interrupt handlers with priority greater than that of I.
B_I	Blocking term for I.
E_I	Upper bound on time required to complete execution of I.
$kernel$	Upper bound on the length of an interval executed with interrupts disabled
$overload_I$	Maximum cost among tasks overloaded with I.

Chapter I
Introduction

1.1 Continuous Media

The wide availability of powerful graphics workstations and low-cost digital audio and video technology has led to the development of *multimedia applications* that integrate audio and video with graphics and traditional data. This integration allows application developers to create revolutionary new tools. However, applications that include digital audio and video data require services not usually found in traditional workstation operating systems. Furthermore, multimedia applications that execute in distributed environments require services not usually provided by traditional networks.

The need for new operating system and network services to support audio and video arises from the continuous nature of these media. Consider video. A real-world scene changes continuously. A digital video camera captures the scene by rapidly acquiring still images called *frames* at a constant rate referred to as the *frame rate*. If frames are acquired at a sufficiently high rate and at regular intervals, and if these frames are displayed at the same rate, then a viewer is presented with the illusion of a continuously changing scene. Digital audio works on a similar principle: sounds are sampled at a very high rate at regular intervals and the samples are played back at the same rate. Media that are acquired and displayed at fixed high rates are known as *continuous media* (CM).

Applications that display CM data must adhere to several timing constraints. First, frames must be displayed at precise intervals. As an example, consider the display of video acquired at the rate of 30 frames a second. To give the illusion of smooth motion, each frame must be displayed for exactly 1/30th of a second. To satisfy this requirement, an application must be able to execute the operations necessary to display frames so as to guarantee that new frames are displayed at specific times. Similar timing constraints exist for the operations that acquire video frames.

A second timing constraint arises when CM applications are used for interactive communication (*e.g.*, a videoconference between geographically separated users). In such cases, the CM data is referred to as *live* continuous media. A key measure of the performance of applications that support live CM is *display latency*. The display latency of live CM data is defined as the elapsed time from acquisition of the data at a source on one workstation to display of the data at a second workstation. Effective communication between users is hampered when display latency is high (*e.g.*, consider the effect of delay in a phone conversation conducted over a satellite link). The timing constraint on CM applications with distributed users is that the display latency must be small enough that the round-trip delay in the users' communication is acceptable.

To meet these timing constraints, a CM application must rely on adequate performance from the underlying network and operating system. Two of the most important performance parameters are bounds on *end-to-end delay* and *end-to-end delay jitter*. To motivate these terms, it is useful to view the process of generating and displaying live CM data as a distributed pipeline. Each frame is generated, undergoes some intermediate processing (*e.g.*, video frames may be compressed), is transmitted over the network, undergoes more intermediate processing, and is displayed.

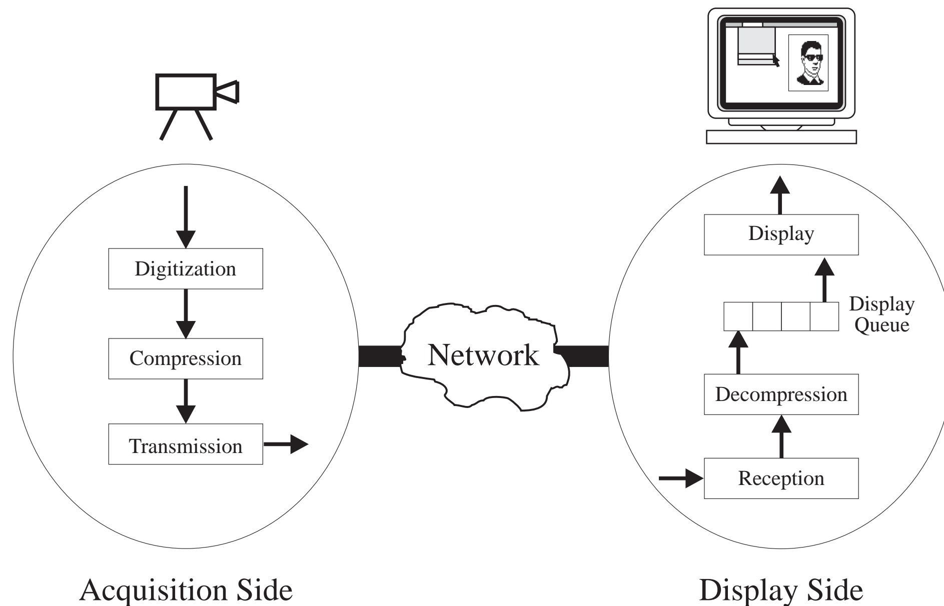

Figure 1-1: A Pipeline View of Continuous Media Processing

Figure 1-1 illustrates this pipeline. Of particular interest is the buffer placed immediately before the display stage of the pipeline. This buffer is referred to as the *display queue* and

is implemented as a queue of individual buffers each of which can hold a single frame. It is necessary for two reasons. First, since frames are generated at one workstation and displayed at another, the processes that generate and display frames are presumably not synchronized. Thus, buffering must exist somewhere in the pipeline to hold frames waiting to be synchronized with the display process. More importantly, unless each stage of the pipeline processes each frame with constant delay, the time required for individual frames to move through the pipeline will vary. Since the display process displays new frames at a fixed rate, variation in the arrival of frames at the display process must be "smoothed out" with buffering. In the idealized pipeline shown in Figure 1-1, the display queue provides any buffering required for frames to synchronize with the display process.

The *end-to-end delay* of a CM frame is defined as the elapsed time between the generation of the frame and its arrival at the display queue. *End-to-end delay jitter* is a measure of the variability in end-to-end delay of frames.

1.2 Delay Jitter

An application that displays live continuous media must address several problems that arise because the end-to-end delays experienced by individual frames can vary. Consider video frames in the pipeline illustrated in Figure 1-1. Initially, frames enter the pipeline at regular intervals of approximately 33 ms. If the delay experienced by each frame at the first stage is constant, then arrivals at the next stage in the pipeline will also occur at regular intervals. However, when the delay at a stage can vary, frames will arrive at the next stage irregularly. As a result, several frames may arrive at the next stage in rapid succession (*e.g.*, several frames arrive at the network interface of the display workstation in a short interval); this is called a *burst*. Because resources such as available processor time and buffer space may be limited, the arrival of a burst can result in loss of frames.

Another problem resulting from delay jitter is that it becomes difficult for an application to display frames "smoothly". Ideally, an application should display each continuous media frame immediately after its predecessor (*i.e.*, frame N+1 should be displayed immediately after frame N). However, if the end-to-end delays experienced by frames vary, then this is not always possible. For example, consider a case where a frame incurs a particularly long end-to-end delay. As a result, the frame may not be available when the display of the preceding frame is complete and the application will be unable to display the new frame.

Such an event is called a *gap* in the display. In the application illustrated in Figure 1-1, a gap occurs whenever the display queue is empty when the display of a frame completes.

Delay jitter can also lead to increases in display latency. To understand why, it is instructive to consider the display queue in Figure 1-1 from the perspective of queuing theory. Assuming no loss, frames arrive at the display queue at an average rate equal to the rate at which frames are generated (*i.e.*, the frame rate). However, because delays experienced by frames in the pipeline can vary, the interarrival time can vary. Thus, the arrival process at the display queue has a general distribution with a mean equal to the frame rate. On the other hand, frames are removed from the display queue (to be played) at periodic intervals defined by the frame rate. Thus the service process for the display queue is deterministic with a mean equal to the frame rate. Queuing theory tells us that, unless the arrival process is deterministic, this queue is unstable. That is, if the end-to-end delays experienced by frames can vary, and if all frames are assumed to arrive, then the length of the display queue can grow without bound. The implication of this observation for applications that display continuous media is that if frames are reliably delivered, then in the presence of unbounded delay jitter the display queue will grow longer over time. As a result, frames will wait longer in the display queue, and thus display latency will grow over time.

Overall, the effect of delay jitter on the display of continuous media frames can be broken into three potential problems:

- Bursts cause loss of frames.
- Large variation in end-to-end delay causes gaps.
- Growth of the display queue causes high display latency.

Furthermore, under the natural assumption that small variations in delay are more common than large variations in delay, there is a tradeoff between minimizing display latency and minimizing gap frequency. The tradeoff results from the fact that the shorter the display queue, and thus the lower the display latency, the higher the probability of encountering an end-to-end delay sufficient to cause a gap.

1.3 Research Approach and Contributions

The goal of my research is an understanding of the fundamental principles governing the processing and display of continuous media in the presence of delay jitter encountered in distributed systems. My approach to this research is to address a particular driving problem: how to support workstation-based videoconferencing applications (*i.e.*, applications that acquire, transmit, and display live audio and video data) in an environment consisting of today's personal workstations, today's commercially available audio/video hardware, and today's networks (*e.g.*, Ethernets, token rings, etc.). There are four principal reasons for studying this problem. First, there is commercial demand for workstation-based videoconferencing systems based on commonly available hardware. Second, while long-haul network providers may soon support communication services with low delay and low delay jitter, today's installed network base will likely continue to be used to support communication within buildings. Third, solutions to the problems of supporting live audio and video data can be applied to a larger class of continuous media data types (*e.g.*, moving images generated for display in virtual reality applications). Finally, solutions for today's environment can be used to evaluate the costs and benefits of specialized services for audio and video that will appear in next generation workstations, audio/video hardware, and networks.

For the purposes of this dissertation, I have imposed two additional constraints on the driving problem. First, I will only address solutions based on end-to-end network transport protocols (*i.e.*, the network is treated as a "black box"). This constraint arises from the observation that for the foreseeable future, it is desirable that audio and video capable workstations can operate without requiring changes to existing network infrastructure (including the software at gateways and bridges). The second constraint is that I will only address transport protocols that operate without feedback from the destination to the source. Such protocols are desirable if audio and video data are broadcast to many destinations from a single source.

To address the driving problem, I have constructed a CM application that acquires video data from a camera attached to a workstation, transmits it over a network, and displays it on the monitor of a second workstation. In addition, the application also acquires audio data from a microphone attached to the first workstation and plays it on speakers attached to the second workstation. The workstations are 66 MHz IBM PS/2 personal computers based on the Intel 486 microprocessor. Each workstation is outfitted with IBM-Intel

ActionMedia 750 adapters for acquiring, compressing, decompressing, and displaying audio and video. In addition, each workstation contains an IBM 16/4 Token Ring or an IBM Ethernet adapter. The workstations are connected through a *campus-sized network*, defined as an internetwork consisting of several local-area networks connected by bridges and routers. The primary hardware environment is illustrated in Figure 1-2.

Because processing and network transmission delays in this environment can vary, a key question that must be addressed is how can the effect of delay jitter on the display of live audio and video be minimized? I propose a two part approach to address this question.

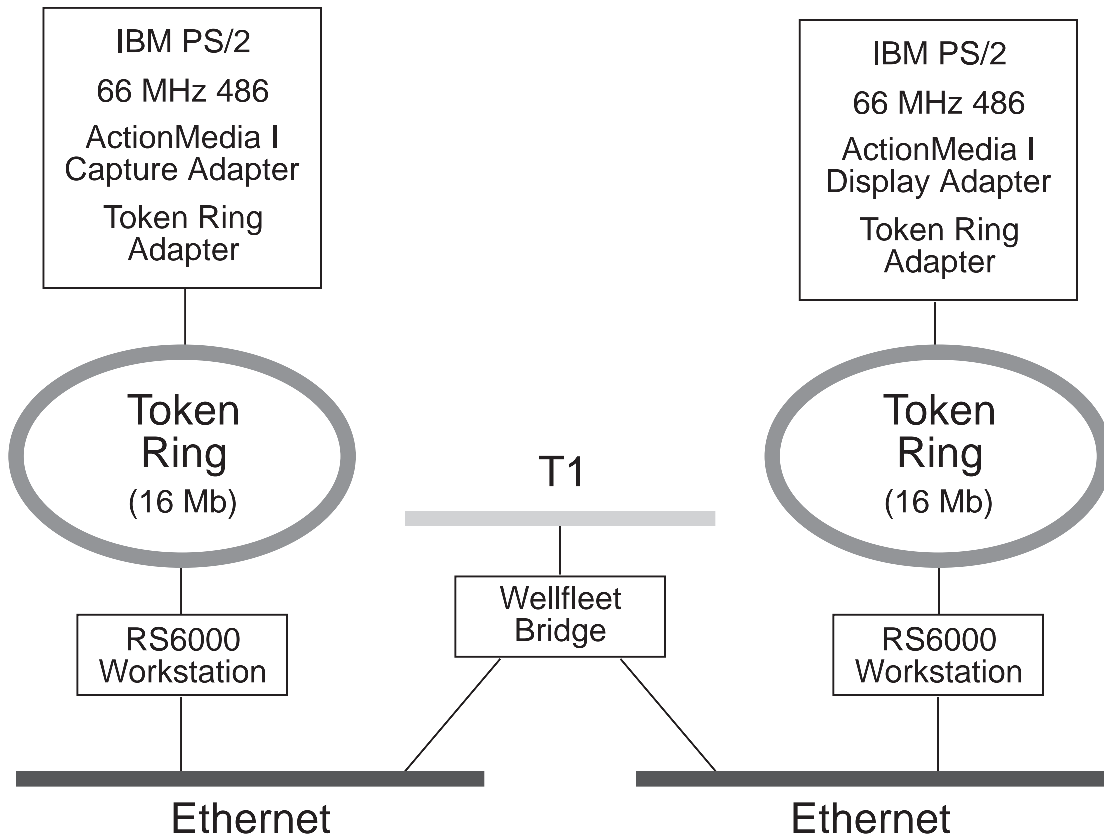

Figure 1-2: Hardware Environment

First, I bound delay jitter at the source and destination workstations; thus under my assumption that the network is a black box, I bound that portion of delay jitter that I can control. This is achieved by designing, analyzing, and implementing the software as a real-time system with strict performance requirements. As a result, I can show that end-to-end delay and end-to-end delay jitter, excepting those delays due to network transmission, are tightly bounded. The second part of my approach is to use adaptive best-effort techniques to account for delay jitter in the network. Network delay jitter is accounted for by

managing the display queue with a policy called *queue monitoring* that dynamically adjusts display latency to accommodate observed delay jitter. Thus, the thesis of this dissertation is that:

> The variation in delays encountered when transmitting continuous media over a campus-sized network is large enough that it must be explicitly addressed in the design of distributed live continuous media applications. A sufficient approach is to combine real-time design, analysis, and implementation techniques to control delay jitter in end-systems with best-effort techniques for managing the effect of delay jitter in the network.

This dissertation will make contributions in several areas. In the area of real-time systems, it will expand the toolkit of scheduling theory and analysis techniques available to the designers of hard-real-time systems and provide a case-study of the design, analysis, and implementation of a significant real-time system. In the area of network and operating system support for continuous media, the dissertation will introduce and evaluate a policy for ameliorating the effect of delay jitter on the display of continuous media frames, provide data on the delay jitter that is experienced by continuous media data in campus-sized networks, and provide a case study of the design of a continuous media application in an environment consisting of today's personal workstations, today's commercially available audio/video hardware, and today's networks.

1.4 Related Work

A number of products and research efforts in both industry and academia have addressed the problem of supporting continuous media applications in the presence of delay jitter. Approaches to the problem can be broken into two categories: those that reduce or eliminate delay jitter and those that accommodate delay jitter. Approaches to reducing or eliminating delay jitter can be further divided into those approaches that reduce delay jitter on the network, and those that reduce delay jitter at the endpoint workstations. All of these approaches are complementary; if delay jitter can be bounded or eliminated in any stage of the processing of continuous media frames, then it becomes easier to accommodate the remaining delay jitter. In this section, I describe a number of products and research efforts that have used one or more of these approaches.

1.4.1 Approaches that Reduce Delay Jitter in the Network

Ideally, the network used by a distributed continuous media application would provide transmission with a guarantee of low delay and low delay jitter. Ferrari gives a good overview of general requirements for real-time communication services including transmission of continuous media [6,7]. Among these requirements are bounds on delay and delay jitter. Ferrari describes two useful classes of such bounds: a deterministic bound is a guarantee that delay (or delay jitter) will be less than the bound, while a statistical bound is a guarantee that the probability that delay (or delay jitter) will exceed some threshold is less than the bound. In addition, he proposes a general scheme for implementing bounds on delay jitter [8]. Such bounds are among those commonly referred to as *Quality of Service* (QOS) guarantees. A good survey of networks and protocols for supporting general QOS guarantees is given in [5]. Here, I will highlight a few of these networks and protocols that support QOS guarantees on delay and delay jitter.

A straightforward way of supporting guarantees on delay and delay jitter is to use a dedicated transmission line (*e.g.*, a T1 connection). This is the approach that has been used in a number of room-based videoconferencing systems. A similar but less expensive approach is to use ISDN services that provide low delay and low delay jitter at a lower bandwidth than dedicated lines. Intel's ProShare is an example of a commercial workstation-based videoconferencing product based on ISDN [19].

Next generation high-bandwidth network technologies such as ATM (Asynchronous Transfer Mode) [35,30] and FDDI (Fiber Distributed Data Interface) [43,29] have been explicitly designed to support the transmission of high-bandwidth, fixed-rate data such as continuous media with QOS guarantees alongside traditional data types with bursty transmission rates. The Pandora system is an example of a system that supports continuous media using an ATM network [28,16].

Work has also been done on the problem of supporting QOS guarantees using networks that were not originally designed to support such guarantees. This work has generally been based on the principle of *resource reservation*. In this approach, applications that wish to transmit data over the network specify the traffic they wish to send, and their desired QOS guarantees, and the network responds by reserving sufficient processor capacity, buffer space, etc., at each hop in the network to ensure that the application receives the desired service. A good discussion of principles used in this approach and an

example of protocols that embody this approach is given in an overview of the Tenet project [7].

Another project that has used resource reservation extensively is the DASH project from Berkeley [1]. The early work on this project included both formal and systems components. The formal aspect of the project was the definition of the DASH Resource Model to describe the resources required by applications that support continuous media. In this model, every device and software component that handles CM is considered a resource. To manage the network resources, the DASH project developed the *Session Reservation Protocol* (*SRP*). SRP operates by allowing applications to reserve capacity at each host in an IP internetwork, and then use standard IP protocols to transmit data [2].

Another protocol based on resource reservation for adding QOS guarantees to IP networks is the *ST-II* protocol [52]. An implementation of this protocol for Token Ring networks was developed by researchers at the IBM European Networking Center as the foundation of the *Heidelberg Transport System* (*HeiTS*), an end-to-end communication system for continuous media data as well as traditional data [13,14,15].

1.4.2 Approaches that Reduce Delay Jitter at the Endpoint Workstations

In traditional workstation operating systems such as Unix, processes can experience a wide range of delays. If processes are used to generate, process, or display continuous media frames, then those frames will experience a high level of delay jitter at the endpoint workstations. Thus, workstation operating systems such as Unix do not provide a good base for building continuous media applications. For example, in his description of a virtual reality system that displays images in a head-mounted display, Azuma notes that the variable delays experienced by processes running under Unix lead to unacceptably large errors in the correspondence between objects in the virtual world and objects in the real world [3].

One approach that has been used to address the problems encountered when using traditional workstation operating systems to support continuous media is to implement critical continuous media functions with high-priority processes. This is the approach used by the HeiTS project in which continuous media applications were implemented using high-priority threads on PS/2 workstations running OS/2 [33].

However, in [38], Nieh, et al. show that the addition of a “real-time class” of processes in Unix SVR4 (*i.e.*, a class of processes that execute with higher priority than any other processes) is not sufficient to allow continuous media applications to effectively coexist with other applications. In particular, graphical user interfaces and other applications that require quick response time do not perform well in the presence of a continuous media application executing at high priority.

Other approaches have attempted to integrate real-time processes more carefully into workstation operating systems. In [9], Fisher describes experiences with a set of modifications to the Unix kernel to support better response times for processes in a real-time class. The ARTs project has used an extension of the Mach workstation operating system, Real-Time Mach, as the basis of techniques for ensuring that real-time processes receive guaranteed service [34,51].

The DASH kernel was designed and implemented as a testbed for experimenting with operating system mechanisms and policies specially tailored to the requirements of continuous media. This kernel allows applications to specify their resource requirements using the DASH resource model. In return, the kernel schedules processes in order to meet the requested QOS guarantees. Specialized implementations of interprocess communication and virtual memory supporting the sharing of CM between processes are also integrated into the DASH kernel [1,10].

An alternate approach that can reduce or eliminate delay jitter at endpoint workstations is the use of special-purpose devices. This was the approach adopted in the Etherphone project at Xerox PARC; basic audio acquisition and playout was provided by special-purpose telephones that digitized, packetized, and transmitted audio data directly onto an Ethernet [50]. Another example of this approach is provided by the Pandora project in which a special purpose device attached directly to the network handles audio and video processing [16,28].

1.4.3 Approaches that Accommodate Delay Jitter

If it is not possible to eliminate delay jitter, or if it is too expensive to eliminate delay jitter, then applications will need to accommodate delay jitter when displaying continuous media frames. Applications accommodate delay jitter by choosing a target display latency large enough that most frames arrive in time to be played, and by attempting to play each frame at that latency. Three issues must be addressed: how does an application estimate the

end-to-end delay experienced by a frame, how does an application choose a target display latency, and at what points should an application choose a new target display latency?

In order to display frames at a target display latency, an application must be able to estimate the end-to-end delay experienced by each frame. Montgomery describes several approaches to this problem [36]. If the clocks at the sender and receiver are synchronized, then the delay experienced by a frame can be determined through the use of timestamps; Montgomery calls this *absolute delay*. If the clocks are not synchronized, then the delay experienced by a frame can be estimated using one of several approaches that are similar to clock synchronization protocols. In each of these approaches, the estimate of the end-to-end delay experienced by a frame is used to determine the time the frame should be held in the display queue before it is played. For example, assume that the target display latency is D, and a frame arrives at the receiver at time t with a delay estimated to be d. If $d \leq D$, then the frame is played at time $t+D-d$. Otherwise, the frame is late and must either be discarded, or played at a latency higher than the target display latency.

Alternately, a conservative assumption can be used in place of an accurate estimate of end-to-end delay. In this approach, which Montgomery calls *blind delay*, the receiver assumes that the first received frame experienced minimum possible delay and delays the display of the frame accordingly (*e.g.*, if the minimum possible end-to-end delay is assumed to be d, the target display latency is D, and the first frame arrives at time t, then the receiver plays the first frame at time $t+D-d$). Then, each successive frame is displayed immediately after its predecessor (*i.e.*, with the same display latency as the first frame).

The Internet Engineering Task Force (the IETF) has used Montgomery's classification of delay estimation techniques in their work on practical solutions to the problems of supporting continuous media in the Internet. In [45], Schulzerinne includes a discussion of these techniques in his discussion of the requirements for *RTP* (the Real-Time Transport Protocol), the IETF's transport protocol for continuous media [46]. Because it results in the smallest error in the estimate of delay, Schulzerinne recommends the use of absolute delay. Nevertheless, in environments in which it is undesirable (or impossible) to synchronize clocks, blind delay is a useful technique.

The problems of choosing a target display latency and determining when to choose a new target have been studied primarily in the context of applications that support audio. In many of these applications, audio is modeled as of a sequence of "talkspurts" (some period of time in which audio data must be acquired, transmitted and played) separated by

"silent periods" (some period of time in which there is no significant audio activity, so audio need not be acquired or played). In [37], Naylor and Kleinrock proposed that a display latency be chosen at the beginning of each talkspurt by observing the transmission delays of the last *m* audio fragments, discarding the *k* largest delays, and choosing the greatest remaining delay. For their particular model of audio quality, they stated a rule of thumb for choosing *m* and *k* ($m > 40$ and $k = .07*m$) that usually resulted in good quality audio. Nevot provides a more recent example of an application that chooses a new display latency at the beginning of each talkspurt based on observations of recent delay jitter [44].

In some sense, the start of a talkspurt provides a convenient opportunity to choose a new target display latency, since display latency can be changed simply by shortening or extending the length of a silent period. However, such convenient opportunities do not necessarily exist for continuous media data types other than audio. Furthermore, talkspurts in audio data other than speech (*e.g.*, music) may be quite long, resulting in few opportunities to change display latency. In such a case, another mechanism must be used to determine when the target display latency should be changed. One example is provided by the clawback buffer mechanism in the Pandora system [28]; display latency is reduced when the display queue has contained more than a target amount of audio for a sufficiently long interval (the clawback buffer is discussed in more detail in Chapter 6).

1.4.4 Summary

In the remainder of the dissertation, I will address the problem of reducing delay jitter at the endpoint workstations, and the problem of accommodating delay jitter. My approach to reducing delay jitter at the endpoint workstations is to use a combination of a real-time operating system and formal modeling and analysis techniques to support the implementation and performance analysis of continuous media applications. By using a real-time operating system to support continuous media, I am taking a similar approach to those used by HeiTS and ARTS; however, in this work I emphasize the formal analysis of the real-time system to determine hard bounds on the delay and delay jitter experienced by continuous media. In contrast to DASH which uses mechanisms designed to directly supporting their formal model of continuous media, I am addressing the use of operating system mechanisms designed for general real-time systems to support continuous media. My approach to accommodating delay jitter is to use a generalization of the policy used to manage clawback buffers in Pandora.

I will not, however, address the problem of reducing delay jitter in the network. There are two reasons. First, guaranteed bounds on delay and delay jitter are not provided by the network hardware or network protocols that I wish to support. Second, since I am only addressing end-to-end solutions, I am unable to use a resource reservation approach. Nevertheless, my approach is complementary to these approaches; anything done to reduce delay jitter in the network will result in less delay jitter to be accommodated at the display.

1.5 Dissertation Overview

The centerpiece of this dissertation is a prototype system for acquiring, transmitting and displaying audio and video. Chapter 2 provides a detailed description of this system. It begins with a discussion of YARTOS, a real-time operating system kernel that runs on the acquisition and display workstations (see Figure 1-2) and supports a real-time programming model in which interrupt handlers, operating system services, and application code execute to completion before well-defined deadlines. Next, the programming interface to the audio/video hardware is described. This description includes pseudo-code for the set of YARTOS tasks used to control the acquisition, compression, decompression, and display processes. Finally, the programming interface to the network is described along with the tasks that control the transmission and reception processes.

Chapters 3, 4, and 5 present a performance analysis of the application. The objective in these chapters is to demonstrate that audio and video frames are processed at the acquisition and display machines with bounded delay. Because the analysis of audio is similar to that for video, and the analysis for the display-side is similar to that of the acquisition-side, I concentrate on showing simply that video frames are acquired and processed on the acquisition-side with bounded delay.

In Chapter 3, I define an abstract model of real-time systems that is implementable using the programming model of YARTOS; for this model, I develop conditions that are sufficient to show that application tasks can be guaranteed to execute prior to application-defined deadlines. In Chapter 4, these conditions are shown to hold when the acquisition-side of the application is defined in terms of the abstract model. In Chapter 5, the fact that each task will always execute prior to its deadline is included as an axiom in an axiomatic specification of the software and hardware on the acquisition machine; then the fact that delay at the acquisition machine is bounded is derived from this specification.

Chapters 6 and 7 discuss and evaluate best-effort policies for accommodating delay jitter in the network. Chapter 6 describes several policies for managing delay jitter. Chapter 7 evaluates these policies with an empirical study performed using the prototype system.

Finally, Chapter 8 presents a summary of the dissertation and my conclusions. The real-time implementation of the system, along with the best-effort mechanisms for accommodating delay jitter in the network are shown to be sufficient to provide acceptable display of audio and video data transmitted over campus-sized LANs.

Chapter II
System Description

2.1 Introduction

The thesis of this dissertation is that the combination of best-effort techniques for managing delay jitter in the network with real-time design and implementation techniques to control delay jitter in end-systems is sufficient to support distributed live continuous media applications in a building-sized network. To evaluate this thesis, I have constructed a workstation-based videoconferencing application that acquires audio and video at one workstation, transfers it over a network, and displays it at a second workstation. The purpose of this chapter is to describe this application. In particular, the description includes the implementation details needed to develop the performance analysis of the acquisition-side of the application described in Chapters 3, 4, and 5.

The design and implementation of the application is based on an operational understanding of several hardware devices and their associated device drivers. Unfortunately, I do not have access to documentation for either the hardware interfaces or the source code for the device drivers used by the application. Instead, I have used several other sources of information to gain an understanding of the low-level behavior of the hardware and device drivers. These include clues derived from documentation for user-level libraries [17, 18, 20], information provided by authors of proprietary software [49], and empirical study of executing applications. Thus, while the descriptions of the hardware interfaces in this chapter are sufficient for understanding the design and implementation of the application, they may be incomplete in some details.

Section 2.2 provides a high-level description of the application and the mechanics of acquiring, transmitting, and displaying audio and video frames. Section 2.3 discusses the handling of hardware interrupts on PS/2 workstations. Section 2.4 describes YARTOS, the operating system kernel on which the application executes. Section 2.5 describes the acquisition-side of the application (*i.e.*, that portion of the application that runs on the workstation that is connected to the camera and microphone); the process of acquiring

and compressing audio and video frames is described along with the interrupt handlers, application tasks, and resources that perform this process.

2.2 Overview of the Application

The basic function of the experimental application is to acquire audio and video data at a workstation, transmit it over a network, and display it at a second workstation. Both sides of the application run on 66 MHz IBM PS/2 workstations based on the Intel 486 microprocessor. The workstations typically communicate through an internetwork of ethernet and token ring networks running the IP protocols. Each workstation is connected to this network through an IBM 16/4 Token Ring adapter (or an IBM Ethernet adapter). In addition, each workstation is outfitted with IBM-Intel ActionMedia 750 adapters for processing digital audio and video. On the acquisition-side, a set of ActionMedia adapters connect the workstation to a camera and microphone and produce digitized audio and video data. On the display-side, another ActionMedia adapter is used to display digital video on the monitor of the workstation and to play digital audio on attached speakers.

Video frames in the application are full-color still images acquired at a rate of 30 frames per second with a resolution of 256x240 pixels. Each frame is processed in several stages. First, it is acquired and digitized by the ActionMedia hardware. Next, the frame is compressed by the ActionMedia hardware. After compression, the frame is added to the queue of frames waiting to be transmitted on the network. Once the frame is at the head of the queue, it is divided into packets. These packets are then transferred over the network to the display-side. On arrival, the packets are reassembled into a frame, and the frame is added to a queue of frames waiting to be decompressed and displayed. At regular intervals of approximately 33 ms., a frame is removed from this queue and decompressed. Finally, the frame is displayed.

Audio processing in the application differs from video processing in two ways. First, audio data is not compressed. Rather, the audio subsystem of the ActionMedia hardware delivers audio directly to the application at a data rate of 120 Kb per second. Second, there is no fundamental unit of audio data directly analogous to the video frame. Digitized audio data is continually written into an internal hardware buffer, and an application may remove data from this buffer at any time. Nevertheless, in the design of the application, I have chosen to manipulate audio data in atomic units of 1/60th of a second. For

convenience, these atomic units will be called audio frames. Thus, in the application, audio data consists of frames that are acquired and displayed at regular intervals of 1/60th of a second.

The stages of processing audio are similar to those for video. First, audio data is acquired and digitized by the ActionMedia hardware. Next, a frame of audio is read from the internal audio buffer and added to the queue of audio frames waiting to be transmitted over the network. The frame is then transferred to the display-side and added to the queue of audio frames waiting to be displayed. At regular intervals of approximately 16.5 ms., an audio frame is removed from this queue and played.

2.3 Hardware Interrupts

On the PS/2, devices communicate with the CPU using a combination of interrupts, I/O commands, and memory-mapped I/O. In particular, the CPU communicates with the ActionMedia and network adapters used by the application by passing data to and from the adapters with memory-mapped I/O; these adapters signal events to the CPU with interrupts.

The delivery of interrupts to the CPU is controlled by a pair of Intel 8253 programmable interrupt controllers. Individual devices are assigned to one of 16 *interrupt request lines* (IRQs). When an IRQ is raised, the interrupt controller raises an interrupt on the CPU according to a set of priority rules. For the mode in which the application uses the interrupt controller, each IRQ has a static priority. An interrupt is raised only if no IRQ with a higher priority is currently being serviced. Otherwise, it is delayed until all higher priority interrupts have been serviced.

In addition to the servicing of a higher-priority interrupt, there are two other reasons why an interrupt may be delayed. First, there is a flag on the CPU that disables all interrupts. This flag is used by the YARTOS kernel to enforce critical sections. Second, the 8253 allows an application to mask individual interrupts, a feature used on the display-side by the handler for token ring adapter interrupts.

IRQ number	Device	Interrupt Handler
IRQ 0	PS/2 timer	TIMER
IRQ 9	ActionMedia adapter	DVI
IRQ 10	ActionMedia adapter	DVI2
IRQ 15	Network adapter	NETWORK

Figure 2-1: Table of Hardware Interrupts

During execution of the application, four different hardware interrupts will be encountered. IRQ0 is raised periodically by an Intel 8259 programmable timer at a rate of 18.2 times per second (*i.e.*, every 55 ms.). IRQ9 and IRQ10 are raised by the ActionMedia adapters to signal the application that one of several events has occurred. IRQ15 is raised by the network adapter to signal the application that a network event has occurred. (The events raised by the ActionMedia and network adapters are detailed in Section 2.5.) Figure 2-1 lists these interrupts in priority order (highest to lowest) along with the name of the corresponding interrupt handler.

2.4 YARTOS

Operating system support for the application is provided by an operating system kernel I have developed called YARTOS (Yet Another Real-Time Operating System). This kernel was originally developed to provide low-level support for the construction of real-time systems specified according to a programming discipline called the Real-Time Producer/Consumer (RTP/C) paradigm [23]. Use of the RTP/C paradigm aids a system designer in specifying throughput constraints and showing that a real-time system adheres to these constraints. YARTOS supports the construction of more general real-time systems with both throughput and response time constraints.

In general, YARTOS is designed to support the construction of systems in which software executes in response to events generated by processes external to the system (*e.g.*, interrupts from hardware devices)[1]. In particular, it is designed to support systems in which the time required to respond to an event must be predictable. YARTOS achieves this goal by providing a programming model that is consistent with a formal model of real-time systems (developed in Chapter 3). This programming model allows an application

---

[1]Such systems are often referred to as *reactive systems*.

developer to express a system design in terms of a formal model that supports the use of formal techniques to analyze the real-time response of the system.

2.4.1 Programming Model

In a YARTOS application, software is divided into a set of interrupt handlers and a set of application tasks. Interrupt handlers and application tasks are sequential programs that execute in response to different kinds of events: interrupt handlers execute in response to hardware interrupts and application tasks execute in response to messages generated by interrupt handlers, other application tasks, or YARTOS itself. In all cases, it is assumed that interrupts or messages will be generated repeatedly, with each resulting in one complete execution of a corresponding interrupt handler or application task.

```
handler <name>
interrupt <IRQ>
body
   <sequential program>
end body
```

Figure 2-2: Interrupt Handler Declarations

Before an application may be executed under YARTOS, the set of interrupt handlers and application tasks must be declared[2]. The syntax of an interrupt handler declaration is given in Figure 2-2. There are three components in a declaration: a name, the interrupt the handler responds to, and the sequential program that should be executed each time the interrupt occurs.

```
task <name>
period <time>
deadline <relative deadline>
resources <resource list>
body
   <sequential program>
end body
```

Figure 2-3: Application Task Declarations

[2]The YARTOS programming model presented here uses an abstract syntax. In the actual implementation of YARTOS, interrupt handler and application task declarations are records with fields corresponding to each component of the abstract declaration, and the sequential programs are functions written in the C language.

The syntax of an application task declaration is given in Figure 2-3. There are several components to this declaration: a name, a relative deadline, a list of resources, and the sequential program that should be executed each time the application task is invoked. In addition, the declaration may optionally specify a period at which YARTOS should send messages to the task. These components of the declaration are discussed in turn.

One component of an application task declaration is a *relative deadline*. YARTOS is designed to ensure that each invocation of an application task executes to completion within an interval beginning at the time the task is invoked and ending at a deadline. The length of this interval is defined as the relative deadline of the task (*e.g.*, each invocation of a task with a relative deadline of 10 ms. is supposed to complete execution within 10 ms. after the task is invoked).

Another component of an application task declaration is a list of *resources*. A resource is an abstraction provided by YARTOS to allow application tasks to share data. Syntactically, a resource is simply a symbolic name. The list of resources in the task declaration is the set of resources "used" by the task. YARTOS guarantees that tasks that use the same resource are granted mutually exclusive access to that resource. Mutual exclusion is maintained by prohibiting tasks that share a resource from preempting one another.

An optional component of an application task declaration is a *period*. Most application tasks are invoked when they receive a message sent by an interrupt handler or another application task using the YARTOS `send_message` system call. However, if an application task is declared with a period, the YARTOS kernel periodically sends messages directly to the task (*i.e.*, if a task is declared with a period of 10 ms., YARTOS will send a message to the task every 10 ms.).

2.4.2 YARTOS System Calls

YARTOS supports three system calls. Declarations of these calls are given in Figure 2-4. The first call is `create_application`. This call takes a set of interrupt handler and application task declarations as an argument. In response, the YARTOS kernel creates the interrupt handlers and application tasks and binds the interrupt handlers to the hardware interrupts.

```
procedure create_application(s: set of declarations)
procedure send_message(t: application_task);
function eventcount(t: task) returns integer;
```

Figure 2-4: YARTOS System Calls

The next system call is `send_message`. This call is used by either interrupt handlers or application tasks to invoke a task. Whenever a message is sent to an application task, the YARTOS kernel creates a new thread of control in which to execute the task. This thread, called a *task invocation*, is assigned a deadline and added to a list of ready tasks. YARTOS schedules ready task invocations using an Earliest Deadline First (EDF) discipline (defined in Section 3.4).

Tasks may often wish to perform processing that is conditional on a particular event having already occurred (*e.g.*, transmit a packet only if the previous transmit has completed). If a task determines if the event has occurred by checking a flag set by the task that executes in response to the event, then the evaluation of the conditional will depend on the order in which tasks are scheduled. To allow tasks to reliably determine if an event has occurred independent of the order in which tasks are scheduled, YARTOS provides the `eventcount` system call. This call returns a count of the number of requests for execution of the task or handler. This allows a task to determine if an event has occurred, even though the task that responds to the event may not have executed.

2.4.3 Assigning Relative Deadlines

YARTOS allows an application task declaration to specify an arbitrary relative deadline. However, it is useful to describe some practical guidelines for choosing these deadlines.

One reason for assigning a particular relative deadline to a task is that it performs processing that is subject to some external timing constraint (*e.g.*, a device must be serviced within a short interval). I will refer to relative deadlines imposed by such constraints as *required deadlines*.

If a task does not have a required deadline, then some other rule must be used to choose the relative deadline. A good choice is the *natural deadline* of the task. Assume that the invocations of a task are always separated by at least p time units; in this case, I will define the natural deadline of the task as p. The effect of assigning the relative deadline of the task to be the natural deadline is that each invocation of the task will complete execution prior to the next invocation. Throughout this work, in the absence of a required deadline

or some other constraint on the choice of a deadline, I will choose to declare application tasks with a relative deadline that approximates the natural deadline.

2.4.4 An Example YARTOS Application

I will now present an example application to illustrate the YARTOS programming model. The example is a simple application that counts keystrokes and prints a message with the current count approximately once per second. There are two hardware interrupts used in this example: IRQ0 is a timer interrupt that occurs approximately 18 times per second, and IRQ1 is an interrupt that occurs on each keystroke. Overall, the example application includes three tasks, two interrupt handlers, and one resource.

Typically, a YARTOS application includes an interrupt handler and a corresponding application task for each hardware interrupt. In an application with this structure, the only activity performed by the interrupt handler is to send a message to the task; the task contains the bulk of the code that should execute in response to the interrupt. This task may then send messages to other tasks. This is the structure used in this example.

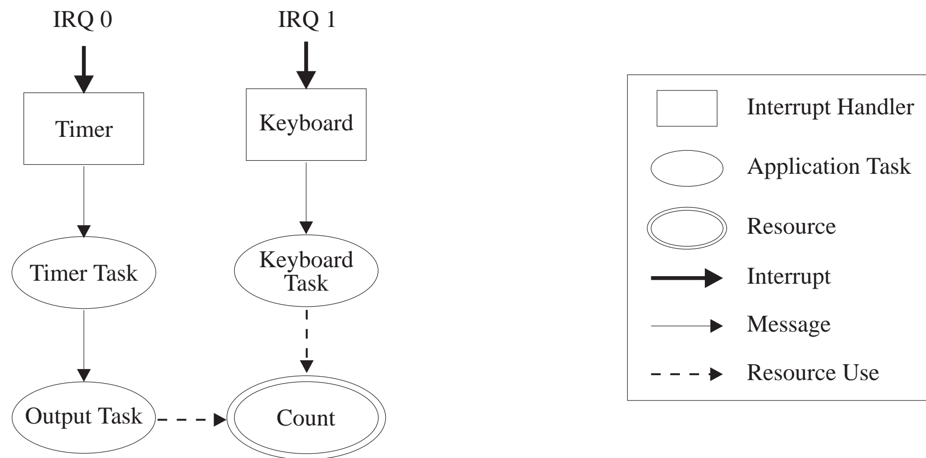

Figure 2-5: Architecture of Example YARTOS Application

Figure 2-5 illustrates the software architecture of this application. Rectangles denote hardware interrupt handlers, single ovals denote application tasks, double ovals denote resources, and arrows from handlers to tasks denote messages sent in response to logical interrupts. Messages from one task to another are also indicated by arrows. Resource usage by an application task (*i.e.*, access to a shared variable) is indicated by a dashed arrow from the task to the resource.

```
Var
   ticks       : integer := 0;         -- count of timer interrupts
   count       : integer := 0;         -- keystroke count

-- Interrupt handler for the timer interrupt
handler timer
interrupt IRQ0
body
   send_message(timer_task);
end body

-- Application task that responds to timer interrupts
task timer_task
deadline 55 ms       -- the natural deadline
resources none
body
   ticks := ticks + 1;
   if ticks mod 18 = 0 then
      send_message(output_task);
   end if;
end body

-- Application task that prints message
task output_task
deadline 1000 ms           -- the natural deadline
resources count
body
   print count;
end body

-- Interrupt handler for the keyboard interrupt
handler keyboard
interrupt IRQ1
body
   send_message(keyboard_task);
end body

-- Application task that counts keystrokes
task keyboard_task
deadline 20 ms             -- a lower-bound on the natural deadline
resources count
body
   count := count + 1;
end body
```

Figure 2-6: Example YARTOS Application

Figure 2-6 lists pseudo-code for the application. It begins with declarations for two global variables, `ticks` which is used to count timer interrupts, and `count` which is used to count keystrokes. `Ticks` is only accessed by one task, but `count` is accessed by two tasks. As a result, in order to ensure that tasks access `count` in a mutually exclusive

manner, each task that uses `count` must include it on the list of resources in the task's declaration.

The next declaration is for the `timer` interrupt handler. This handler is executed whenever the IRQ0 interrupt occurs. It simply uses the YARTOS system call `send_message` to send a message to the application task `timer_task`.

`Timer_task` is an application task that performs all the "real" processing that should be done in response to a timer interrupt. In this application, there is no required deadline for this task, so the relative deadline is set to its natural deadline of 55 ms. which is the expected time between timer interrupts. The body of the task counts the number of times it has executed; every 18 times (*i.e.*, approximately once per second) it sends a message to the application task `output_task`.

`Output_task` is the application task that prints the current keystroke count. Again, this task has no required deadline, so its relative deadline is set to its natural deadline of 1000 ms. The body of the task simply prints the current value of `count`; since this is a global variable shared with another application task (*i.e.*, `keyboard_task`), `count` is listed a resource used by `output_task`.

The next declaration is for the `keyboard` interrupt handler. This handler is executed whenever the IRQ1 interrupt occurs. As with the `timer` handler, it simply sends a message to the `keyboard_task`, an application task that will perform all the "real" processing that should be done in response to a keyboard interrupt.

The final declaration is for the `keyboard_task` application task. While the other tasks had an obvious natural deadline, the natural deadline of this task is not obvious because the minimum time between two keyboard interrupts is not well-defined. Nevertheless, a reasonable lower bound can be estimated; in this case, the relative deadline of the task is set to an arbitrary value of 20 ms. (*i.e.*, 50 keystrokes per sec.). In addition, since the global variable `count` is used by the task, it is included as a resource in the declaration.

2.4.5 Implementation Details

In order to correctly specify the behavior of YARTOS tasks, etc., in the axiomatic specification presented in Chapter 5, it is necessary to discuss two additional implementation details of YARTOS. The first issue is the method by which the deadline of application task invocations is computed. Specifically, the deadline of a task invocation

is defined to be the *logical arrival time* of the invocation plus the relative deadline of the task.

The logical arrival time of a task invocation is defined differently for interrupt handlers and application tasks. The logical arrival time of an interrupt handler is determined by checking the current time; this is the first activity performed when a hardware interrupt occurs. Thus, the logical arrival time of a task is somewhat greater than its actual arrival time. The logical arrival time of an application task invocation is defined to be the logical arrival time of the interrupt handler or application task that sent it a message; that is, when a task is invoked by a `send_message` system call, the logical arrival time of new invocation is set to the logical arrival time of the sender.

The other implementation detail that must be discussed is the method by which YARTOS generates messages to application tasks that specified a period as part of the task declaration. Abstractly, the YARTOS kernel should generate messages to such a task at regular intervals. However, because application tasks can specify an arbitrary period, an ideal implementation of this abstraction would require a clock that could interrupt the processor at arbitrary intervals. In the implementation of YARTOS, I have chosen not to rely on the presence of such a clock.

Instead, the YARTOS kernel approximates the periodic generation of messages with the following technique. For each task with a specified period, YARTOS keeps track of the times at which messages to the task should be generated. Whenever any application task or interrupt handler completes execution, or when the processor is idle, YARTOS checks to see if such a time has passed; if so, it sends a message to the appropriate task. In any case, the logical arrival time of the task invocation is set to the time at which the messages should have been generated (*i.e.*, if the message should have been generated at time *t*, it is assigned a logical arrival time of *t*, even if YARTOS actually generated the message later). The effect of this approximation on the problem of ensuring that application tasks meet their deadline constraints is investigated in Chapter 4.

2.5 Acquisition-Side Processing

This section describes the design and implementation of the portion of the workstation-based videoconferencing application that runs on the acquisition-side workstation (*i.e.*, the workstation that is connected to the camera and the microphone). This portion of the

application does several things: it acquires and compresses video frames, acquires audio frames, and transmits the frames over the network.

The implementation consists of a set of interrupt handlers, application tasks, and resources running on top of the YARTOS kernel. Interrupt handlers execute in response to hardware interrupts and send messages to application tasks. These application tasks perform most of the activities involved in acquiring, compressing, and transmitting audio and video. Application tasks cooperate by communicating data through shared variables; access to these variables is protected with YARTOS resources.

Most of the processing performed by the application is executed in response to the hardware interrupts listed in Figure 2-1. Each hardware interrupt can be raised for one of several reasons. As an example, the IRQ 15 interrupt is raised by the network adapter to indicate that it is ready to accept a new network packet, or to indicate that a packet has been successfully transmitted, or to indicate that a packet has been received. Throughout the remainder of this discussion, I use the term *logical interrupt* to refer to a hardware interrupt raised for a particular reason. The application includes an application task corresponding to each logical interrupt. When a hardware interrupt is raised, the interrupt handler executes, communicates with the hardware to determine which logical interrupt has been raised, and sends a message to the appropriate application task.

I begin by describing some data types and global variables used in the application. Next, I detail the operations the application must perform in order to acquire, compress, and transmit the audio and video frames. I then present a design that divides these operations into a set of YARTOS interrupt handlers and tasks that share data using resources. Finally, I present detailed pseudo-code for the YARTOS tasks.

2.5.1 Basic Declarations

I begin the description of the acquisition-side of the application by defining several data types and primitive operations that will be used in the code. The primary data types used are buffers. There are three types of buffers, defined by the type of data they can hold: a *digitize buffer* can hold one digitized video frame, a *compress buffer* can hold one compressed video frame, and an *audio buffer* can hold one audio frame. Declarations for these three types are listed in Figure 2-7.

```
Type
   digitize_buffer   : array of bytes;
   compress_buffer   : array of bytes;
   audio_buffer      : array of bytes;
```

Figure 2-7: Audio and Video Buffers

Each type of buffer is dynamically allocated from a pool of free buffers of that type; declarations of the memory management routines are listed in Figure 2-8. `Available` and `allocate` are overloaded functions that take a buffer type name as an argument: `available` is a boolean function that returns true if a buffer of the proper type can be allocated from its pool, while `allocate` takes a buffer of the proper type from its pool and returns it. `Free` returns a buffer to the corresponding pool.

```
function available(buffer_type: type) returns boolean;
function allocate(buffer_type: type) returns buffer_type;
procedure free(buffer: buffer_type);
```

Figure 2-8: Memory Management Calls

Figure 2-9 lists declarations for operations used to acquire and compress audio and video frames. `Digitize` initiates a request to the ActionMedia hardware to fill `db` with a new digitized video frame. `Start_compress` initiates a request to the ActionMedia hardware to compress the video frame in `db` and put the result in `cb`. `Audio_acquire` retrieves a new audio frame from the ActionMedia hardware and puts it in `ab`.

```
procedure digitize(db: digitize_buffer);
procedure start_compress(db: digitize_buffer, cb: compress_buffer);
procedure audio_acquire(ab: audio_buffer);
```

Figure 2-9: Audio and Video Operations

Another data type used in the application is the queue. Each queue will contain items of a single type (*e.g.*, a queue of `digitize_buffer` will contain zero or more digitize buffers). Figure 2-10 lists declarations for the operations defined on queues: `length` returns the length of the queue, `insert_queue` inserts an item of the proper type at the tail of a queue, and `remove_queue` removes the buffer at the head of the queue and returns it. In each declaration, `data_type` is a generic name for the type of item contained in `q`.

```
function length(q: queue of data_type) returns integer;
procedure insert_queue(q: queue of data_type, d: data_type);
function remove_queue(q: queue of data_type) returns data_type;
```

Figure 2-10: Operations on Queues

Figure 2-11 lists declarations for the data types and routines used to transmit data over the network. The basic data type is the *packet descriptor*. This is a record that is used to specify the data that should be placed in a network packet. Each packet can contain up to one compressed video frame and up to two audio frames. `Cb_count` is the number of video frames that should be put into the packet, `ab_count` is the number of audio frames that should be put into the packet, and `cb`, `ab1`, and `ab2` are the buffers containing the data that should be put into the packet. The `transmit` routine takes a packet descriptor as an argument and initiates the transmission of the appropriate packet.

```
Type
   packet_descriptor : record
                         cb_count    : integer;
                         cb          : compress_buffer;
                         ab_count    : integer;
                         ab1         : audio_buffer;
                         ab2         : audio_buffer;
                     end record;

procedure transmit(d:  packet_descriptor);
```

Figure 2-11: Network Transmission Declarations

```
Constant
   max_audio_transport:  integer;      -- max buffers "in transport"
   max_video_transport:  integer;      -- max buffers "in transport"

Var
   vbi_count          :  integer;

   next_digitizing    :  queue of digitize_buffer;
   digitizing         :  queue of digitize_buffer;

   compress_source    :  queue of digitize_buffer;
   compress_sink      :  queue of compress_buffer;
   db_freed           :  integer;

   transmit_video     :  queue of compress_buffer;
   video_transport    :  integer;

   transmit_audio     :  queue of audio_buffer;
   audio_transport    :  integer;

   transmit_queue     :  queue of packet_descriptor;
   transmits_started  :  integer;
```

Figure 2-12: Global Variable Declarations

Finally, Figure 2-12 lists a number of constant and global variable declarations. The meaning of the constants and the use of the global variables will be explained below. One general note is in order though: each data structure that holds a buffer during execution is declared as a queue, including those that could have been implemented as a simple variable. This property is reflected in the uniform treatment given the data structures in the axiomatic specification presented in Chapter 5.

2.5.2 High-Level Architecture

The acquisition-side of the application is designed and implemented as a set of YARTOS interrupt handlers and tasks that share data using resources. However, to specify the activities that must be performed, and the timing constraints on those activities, it is useful to first describe the design in terms of higher-level abstract processes. The acquisition-side of the application can be thought of as three concurrent processes: a *video process* that acquires and compresses video frames, an *audio process* that acquires audio frames, and a *transport process* that transmits frames over the network. Note however that these abstract processes do *not* execute at runtime; rather they are presented here in order to give a high-level view of the processing that must be performed by the application. Later in the chapter, it will be shown how the processing described in these abstract processes is realized by a set of YARTOS application tasks.

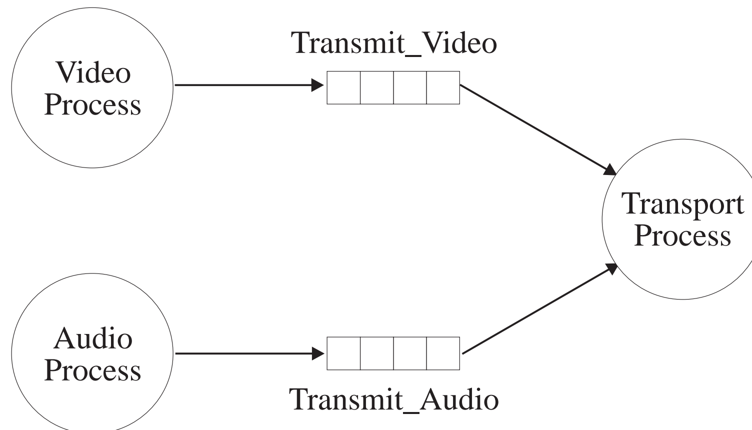

Figure 2-13: High-Level Architecture

The architecture of these abstract processes is illustrated in Figure 2-13. Frames acquired and compressed by the video process are placed in a queue of compress buffers, the `transmit_video` queue. Frames acquired by the audio process are placed in a queue

of audio buffers, the transmit_audio queue. The transport process removes buffers from these queues and transmits the data over the network.

There is one complication in this simple architecture. Because of network congestion, it may not be possible for the transport process to transmit every frame that is generated; if so, buffers will accumulate in the transmit_video and transmit_audio queues. Because the total number of audio and video buffers in the application is limited, this could eventually lead to situations in which buffers are not available to the video process and the audio process. In order to ensure that buffers will be available, I limit the number of buffers defined to be "in the transport system". A buffer is defined to be "in the transport system" if it has been placed on the appropriate transmit queue at some point in the past, and has not yet been freed by the transport process. Two global variables, video_transport and audio_transport, are used to count the number of video and audio buffers in the transport system; each is incremented when a buffer is placed on the appropriate transport queue, and each is decremented when a buffer is freed. Two constants, max_video_transport and max_audio_transport provide bounds on the number of compress buffers and audio buffers respectively. The video and audio processes enforce these limits each time they add a new frame to a transmit queue.

2.5.3 The Video Process

Figure 2-14 shows the abstract video process which acquires and compresses each video frame. At a high level, this process has three steps: a digitize operation to initiate the acquisition of a digitized video frame, a start_compress operation to initiate the compression of the frame, and an insert_queue operation to place the frame on the transmit_video queue. The "WAIT statements" are not executable statements; rather they are placeholders that indicate that further processing should delayed until a particular logical interrupt occurs. Thus the WAIT statements can be thought of as constraints on the timing of these operations. These timing constraints are discussed below.

In the above description, a digitized video frame is acquired by executing the digitize operation. In reality, the acquisition of individual digitized video frames is more complex. The ActionMedia video acquisition hardware continuously acquires, digitizes, and writes video data; the digitize operation merely informs the hardware to begin writing the data to a new location. It takes 1/30th of a second to write the digitized data

corresponding to a single video frame. Thus, the application acquires individual video frames by executing the `digitize` operation at regular intervals of 1/30th of a second.

```
var
   db:  digitize_buffer;    -- holds the digitized frame
   cb:  compress_buffer;    -- holds the compressed frame

WAIT (VBI1);    -- VBI1 signals opportunity to digitize

-- initiate a digitize operation
db := allocate(digitize_buffer);
digitize(db);

WAIT (VBI0);    -- VBI0 signals start of digitize

WAIT (VBI0);    -- 2nd VBI0 signals end of digitize

-- initiate a compress operation
cb := allocate(compress_buffer);
start_compress(db,cb);

WAIT (CC);      -- CC signals end of compress

-- give frame to transport process
insert_queue(transmit_video,cb);
video_transport := video_transport + 1;

-- enforce limit on compress buffers "in transport system"
if video_transport > max_video_transport then
   video_transport := video_transport - 1;
   cb := remove_queue(transmit_video);
   free (cb);
end if

-- free the digitize buffer
free(db);
```

Figure 2-14: High Level View of the Video Process

Specifically, the application acquires video frames by responding to logical interrupts known as *vertical blanking interrupts* (VBI interrupts)[3]. These interrupts are generated

[3]The ActionMedia video hardware is designed to be compatible with the NTSC broadcast television standard. In NTSC, video is scanned in horizontal lines from top to bottom. A complete scan of a video frame occurs in two vertical passes, one for the odd lines of the frame and one for the even lines of the frame. The time during which the scanning point resets to the top of the image is known as the vertical blanking interval. The vertical blanking interrupt is so-named because it occurs at the start of each vertical blanking interval.

periodically by the ActionMedia hardware at a rate of 60 interrupts per second. An application acquires a video frame by executing a `digitize` operation in the interval between two VBI interrupts. At the next VBI interrupt, the hardware will begin writing digitized video into the specified buffer. The application must then wait for two more VBI interrupts before a complete frame has been written to the digitize buffer.

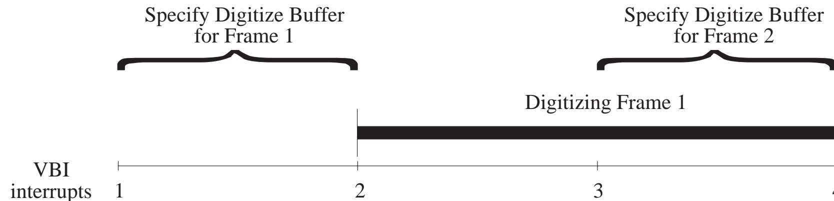

Figure 2-15: Digitization Process

Figure 2-15 illustrates the process of acquiring a digitized video frame. Sometime after VBI interrupt 1 and before VBI interrupt 2, the application must execute a `digitize` operation to pass a digitize buffer to the video subsystem. Between VBI 2 and VBI 4, this buffer is filled with a digitized video frame. After VBI 4, the buffer contains a complete video frame. However, between VBI 3 and VBI 4, the application must execute *another* `digitize` operation. Otherwise, after VBI 4, the video subsystem will continue to write digitized data into the first buffer overwriting the acquired frame.

Thus the first timing constraint indicated by a `WAIT` statement in Figure 2-14 is that `digitize` operations should be executed after every second VBI interrupt; for convenience, I will assume that `digitize` operations are executed after odd-numbered VBI interrupts. Odd-numbered VBI interrupts will be referred to as VBI1 interrupts.

The next timing constraint arises from the fact that the digitized frame has not been completely acquired until the second VBI interrupt after the start of the digitization; the second and third `WAIT` statements in Figure 2-14 indicate that the `start_compress` operation should not be initiated until then. Thus, `start_compress` operations are executed after even-numbered VBI interrupts. These will be referred to as VBI0 interrupts.

The final `WAIT` statement in Figure 2-14 indicates that the compressed video frame should not be delivered to the transport process until the frame has been completely compressed. The `start_compress` operation initiates the compression; a logical interrupt known as the *compress complete* (CC) interrupt is generated by the ActionMedia hardware when the

compression is finished. Thus, the fourth `WAIT` statement indicates that the frame should not be placed on the `transmit_video` queue until the CC interrupt occurs.

2.5.4 The Audio Process

Figure 2-16 lists the statements executed by the audio process to acquire each audio frame. At a high level, this process has two steps: an `audio_acquire` operation to retrieve a digitized audio frame and an `insert_queue` operation to place the frame on the `transmit_audio` queue. The `WAIT` statement indicates a timing constraint dictated by the fact that audio frames are assumed to correspond to a fixed-length interval.

The audio subsystem of the ActionMedia hardware operates by continuously writing digitized audio data to a large internal circular buffer. At any time, an application can copy audio data from this buffer to its own internal memory. The audio subsystem maintains a pointer to the last copied byte, so each copy will begin where the previous copy finished. In the application, audio is acquired using the `audio_acquire` operation that copies 1/60th of a second of audio data (approximately 264 bytes). To ensure that each audio frame is acquired, an `audio_acquire` operation must be executed every 1/60th of a second. Since VBI interrupts are generated at this rate, it is convenient to assume that `audio_acquire` operations should be executed after each VBI interrupt.

```
Var
   ab:  audio_buffer;

WAIT (VBI);    -- VBI signals opportunity to acquire next frame

-- Acquire a new audio frame
ab := allocate(audio_buffer);
acquire_audio(ab);

-- give frame to transport process
insert_queue(transmit_audio,ab);
audio_transport := audio_transport + 1;

-- enforce limit on audio buffers "in transport system"
if audio_transport > max_audio_transport then
   audio_transport := audio_transport - 1;
   ab := remove_queue(transmit_audio);
   free(ab);
end if
```

Figure 2-16: High Level View of the Audio Process

2.5.5 The Transport Process

```
Var
   d:  packet_descriptor;

-- check to ensure all outstanding transmits are completed
if eventcount(TC) < transmits_started then
   return;
end if

-- if available, add video frame to packet
if length(transmit_video) > 0 then
   d.cb_count := 1;
   d.cb := remove_queue(transmit_video);
else
   d.cb_count := 0;
end if

-- if available, add audio frames to packet
if length(transmit_audio) > 1 then
   d.ab_count := 2;
   d.ab1 := remove_queue(transmit_audio);
   d.ab2 := remove_queue(transmit_audio);
else if length(transmit_audio > 0) then
   d.ab_count := 1;
   d.ab1 := remove_queue(transmit_audio);
else
   d.ab_count := 0;
end if

-- initiate transmission, maintain count of transmits initiated
transmits_started := transmits_started + 1;
transmit(d);

WAIT (TC);     -- TC signals end of transmission

-- free the compress buffer, maintain count of buffers "in transport"
if d.cb_count > 0 then
   video_transport := video_transport - 1;
   free(d.cb);
end if;

-- free the audio buffers, maintain count of buffers "in transport"
if d.ab_count > 1 then
   audio_transport := audio_transport - 2;
   free(d.ab1);
   free(d.ab2);
else if d.ab_count > 0 then
   audio_transport := audio_transport - 1;
   free(d.ab1);
end if
```

Figure 2-17: High Level View of the Transport Process

Figure 2-17 lists the statements executed by the transport process to transmit one packet on the network. At a high level, there are four steps in the process. First a check is performed to ensure that all outstanding transmit requests have been completed (described below). If so, then a packet descriptor containing up to one video frame and up to two audio frames is constructed; the frames are removed from the appropriate queue and placed in the packet descriptor. Next, the `transmit` operation is executed to initiate the transmission. Finally, the buffers that were transmitted in the packet are freed.

The `WAIT` statement in Figure 2-17 indicates a timing constraint: the buffers placed in the packet should not be freed until the packet has been successfully transmitted over the network. The `transmit` operation initiates the transmit request; a logical interrupt known as the *transmit complete* (TC) interrupt is generated by the network hardware when the transmission is finished.

The check that ensures all outstanding transmit requests have been completed is based on a YARTOS eventcount of TC interrupts. Each time a `transmit` operation is performed, the `transmits_started` counter is incremented. Then, it is the case that all outstanding transmission requests have completed only if the number of TC interrupts that have occurred is equal to `transmits_started`.

2.5.6 Breakdown into Application Tasks

The next step in describing the acquisition-side of the application is to divide the operations listed in the high-level abstract processes described above into a set of interrupt handlers and application tasks that can execute under YARTOS. Recall that the video process, the audio process, and the transport process were divided into phases separated by `WAIT` statements. These phases defined by `WAIT` statements are the basis of the division of the application into tasks.

With the exception of the first phase of the transport process, which will be discussed separately, each phase begins with a `WAIT` for a particular logical interrupt. Thus, a natural architecture is to define application tasks corresponding to each phase, and arrange for each task to execute in response to the appropriate logical interrupt. For example, consider the fragment of the video process listed in Figure 2-18. The group of statements from the first to the second `WAIT` statements is implemented as one application task. Since the `WAIT` statement that starts the group is for a VBI0 logical interrupt (*i.e.*, an

even-numbered VBI interrupt), this task should be sent a message whenever a VBI0 interrupt occurs.

```
WAIT (VBI0);

cb := allocate(compress_buffer);
start_compress(db,cb);

WAIT (CC);

insert_queue(transmit_video,cb);
```

Figure 2-18: Fragment of the Video Process

A buffer (or other data) that is used in several phases of an abstract process is passed between the corresponding tasks by putting the buffer on a queue. To ensure that access to the queue by each task is mutually exclusive, each task declares the queue as a resource. Again, consider the fragment of the video process listed in Figure 2-18. Figure 2-19 shows this fragment split into two tasks. A queue of compress buffers, `compress_sink`, is used to pass the compress buffer between the two tasks. This queue is included on the resource list of each task.

```
task one
resources compress_sink
body
   cb := allocate(compress_buffer);
   start_compress(db,cb);
   insert_queue(compress_sink,cb);
end body

task two
resources compress_sink
body
   cb := remove_queue(compress_sink);
   insert_queue(transmit_video,cb);
end body
```

Figure 2-19: Video Fragment Divided Into Tasks

Thus, the basic software architecture of the acquisition-side of the application is based on dividing the abstract processes defined in Figures 2-14, 2-16, and 2-17 into application tasks and using queues to pass data between the tasks. In addition to these tasks, the architecture includes an interrupt handler for each of the hardware interrupts listed in Figure 2-1, and several other miscellaneous tasks discussed below.

The overall architecture is illustrated in Figure 2-20. Rectangles denote hardware interrupt handlers, single ovals denote application tasks, double ovals denote resources, and arrows from handlers to tasks denote messages sent in response to logical interrupts. Messages from one task to another are also indicated by arrows. Resource usage by an application task (*i.e.*, access to a shared variable) is indicated by a dashed arrow from the task to the resource.

While the rules described above for dividing the abstract processes into application tasks are sufficient to explain most of the actual implementation, there are several exceptions that must be addressed. First, in the abstract processes listed above, it was assumed that calls to `allocate` always succeeded. In the actual tasks, execution is protected with a call to `available`; if a needed buffer cannot be allocated, the code that allocates and uses the buffer is not executed. Similarly, before a buffer is removed from a queue, the length of the queue is checked to ensure that the `remove_queue` will succeed; if not, the code that requires the buffer is not executed.

Another exception is the location of the code that returns digitize buffers to the free pool. A digitize buffer containing a frame can be returned to the free pool as soon as the frame has been compressed. According to the rules described above, this code should be placed in the `CC` task that contains the code that executes after a CC interrupt signals that a compression is complete. However, because the number of digitize buffers available to the application is restricted by memory limitations on the ActionMedia adapter, this code must execute prior to the next time a digitize buffer is allocated by a `VBI1` task; if not, the pool of digitize buffers will be empty when the `VBI1` task executes (resulting in a lost frame).

It will be shown in Chapter 5 that the CC interrupt signaling that a digitize buffer can be returned to the free pool will always be completed before the buffer must be reused. The problem arises because it cannot be guaranteed that the `CC` task will execute prior to the `VBI1` task that will require the buffer. Thus, the code that frees the digitize buffer is moved from the `CC` task to the beginning of the `VBI1` task and a YARTOS eventcount is used to determine if the CC interrupt has occurred.

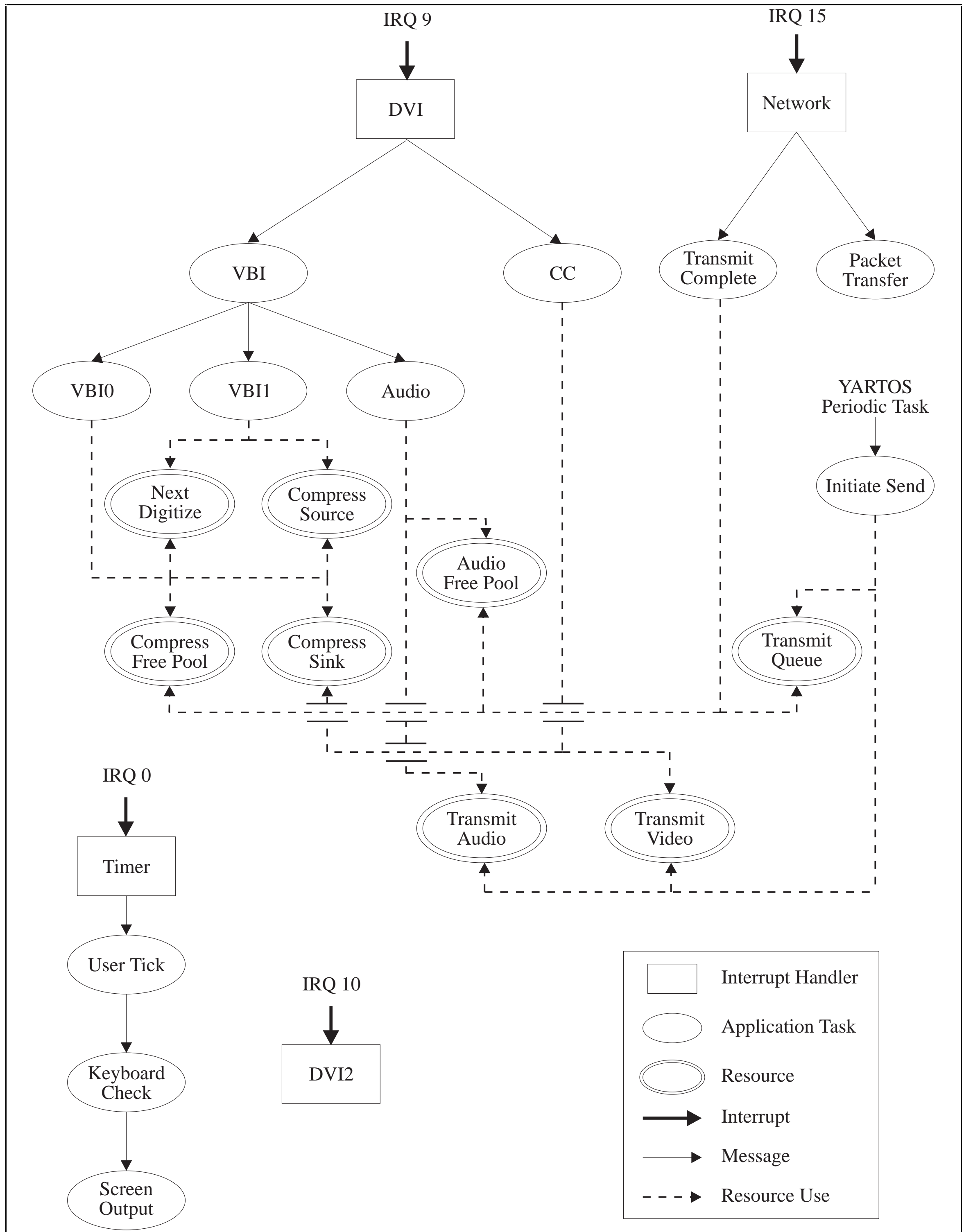

Figure 2-20: Software Architecture of the Acquisition-Side

2.5.7 Assigning Relative Deadlines to the Application Tasks

In choosing relative deadlines for the set of application tasks, there are two constraints that must be addressed. First, each task with a required deadline must be assigned a relative deadline short enough to ensure that the timing constraint is met. Timing constraints are generally based on the actual arrival time of a hardware interrupt; processing must occur within a well-defined interval after the interrupt. But because of measurement delays, and more significantly because the execution of interrupt handlers can be delayed while higher-priority interrupt handlers execute, the logical arrival time of an interrupt handler is somewhat greater than the actual arrival time of the interrupt. Thus, to ensure that a task invoked by an interrupt handler executes within a required interval, it must be assigned a relative deadline somewhat smaller than the timing constraint.

The second constraint that must be addressed when choosing relative deadlines is that it must be possible to schedule the set of application tasks so that each task invocation executes to completion prior to its deadline. This property can be checked using the procedure developed in Chapters 3 and 4.

In addition, for reasons that will be explored further in Chapter 4, any application task that receives a message from another application task should be assigned a relative deadline greater than or equal to the relative deadline of the sender.

The rules I have used to choose relative deadlines for the set of application tasks described here are based on these constraints. On the acquisition-side of the application, there is only one task with a required deadline, the `VBI1` task. Recall that if frames are to be digitized correctly, `digitize` operations must be executed after a VBI1 interrupt and prior to the next VBI0 interrupt. A VBI0 interrupt is expected to occur approximately 16.67 ms. after each VBI1 interrupt. Thus, the `VBI1` task has a required deadline of approximately 16.67 ms. In the declaration of the `VBI1` task, I have chosen to use a conservative estimate of 15 ms. for the relative deadline.

The next task with a constraint on the choice of deadline is the `VBI` task. Because it sends a message to the `VBI1` task, it should be assigned a relative deadline less than or equal to the relative deadline of the `VBI1` task. Thus, I also assign the `VBI` task a relative deadline of 15 ms.

For each of the other application tasks, I have more flexibility in choosing a deadline. For the two other tasks that execute in response to VBI interrupts, `VBI0` and `audio`, I have simply chosen to use the same relative deadline as the `VBI` task, 15 ms. For the `CC` task, I have chosen a deadline of 8 ms. which is a conservative estimate of its natural deadline. For all other application tasks except the `initiate_send` task (discussed below), I have arbitrarily chosen to use a relative deadline of 33 ms.

2.5.8 The Initiate_Send Task

Most of the application tasks discussed here were defined by a group of statements in an abstract process starting with a `WAIT` for a particular logical interrupt. However, because the activities performed by the transport process need not occur in response to a particular logical interrupt, the first phase of the transport process did not begin with a `WAIT` statement (see Figure 2-17). Rather, I have a great deal of flexibility in determining when this code should be executed.

Because one video frame and two audio frames are produced approximately every 33 ms., the application must, on average, transmit one video and two audio frames every 33 ms. The code for the abstract transport process was designed to send one video and two audio frames in a single packet. Thus, this code should be executed at least once every 33 ms. Therefore, this code is placed in an application task called `initiate_send` with a period of 33 ms.[4]

Next, I must assign a relative deadline to this task. As will be discussed below, the execution of a `transmit` operation results in the generation of several logical interrupts by the network hardware. In order to guarantee that application task invocations will always execute prior to their deadlines, I must ensure that successive occurrences of each logical interrupt are separated by a sufficient interval. Thus, I must ensure that `transmit` operations are separated by a sufficient interval. This can be done by ensuring that successive invocations of the `initiate_send` task are adequately separated and that can be achieved by setting the relative deadline of the task to a value

---

[4]Because this task will be invoked at a period of 33 ms., this task could have executed in response to messages generated by alternate executions of the VBI task. However, I chose to use periodic messages generated by YARTOS so that this period could be easily changed.

less than the period of the task. As a result, I have chosen to assign the initiate_send task a relative deadline of 20 ms.

2.5.9 Description of the Interrupt Handlers and Application Tasks

I am now ready to present declarations and pseudo-code for each of the application tasks. Figure 2-21 shows the declaration for the VBI task. This task executes in response to a message generated by the DVI interrupt handler whenever a VBI logical interrupt occurs. The VBI task does not correspond to a phase in one of the abstract processes; rather, it is used to send messages to each of the tasks that execute in response to VBI interrupts.

```
task VBI
deadline 15 ms
resources none;
task body
   vbi_count := vbi_count + 1;

   if vbi_count mod 2 <> 0 then
      send_message(vbi1);
   else
      send_message(vbi0);
   end if

   send_message(audio);
end task
```

Figure 2-21: Pseudo Code for VBI Task

Each time the VBI task executes, it sends a message to the audio task and either the VBI1 or VBI0 task. A count of the number of times the VBI task has executed, stored in the global variable vbi_count, is used to determine if the VBI interrupt is odd or even numbered. The VBI task is assigned a deadline of 15 ms. and does not use any resources (the global variable is not shared with any other task).

The VBI1 task is the application task corresponding to the first phase of the video process (see Figure 2-14). Figure 2-22 shows the task declaration. The relative deadline is set to 15 ms. The resource list includes two global variables that this task shares with other tasks: the next_digitize queue and the compress_source queue. The other two globals used by this task, the pool of free digitize buffers and the db_freed counter, are not used by any other task, and thus need not be included on the resource list. In order to pass the digitize buffer to the next phase, the VBI0 task, code is added to place the buffer on the next_digitize queue.

```
task VBI1
deadline 15 ms
resources next_digitize, compress_source
task body
   var db: digitize_buffer;

   if length(compress_source) > 0 and eventcount(CC) >= db_freed then
      db_freed := db_freed + 1;
      db := remove_queue(compress_source);
      free(db);
   end if

   if available(digitize_buffer) then
      db := allocate(digitize_buffer);
      digitize(db);
      insert_queue(next_digitize,db)
   end if
end task
```

Figure 2-22: Pseudo Code for VBI1 Task

In the abstract video process listed in Figure 2-14, there are two `WAIT` statements for the VBI0 logical interrupt. The two waits ensure that a compress operation is not started on a video frame until the digitization is complete. This property must also be ensured in the actual implementation.

Each time the `VBI0` task executes, it does two things. First, it removes a digitize buffer from the `digitizing` queue and performs the activities listed in the second phase of the video process (*i.e.*, initiate a `compress` operation, etc.). Second, the `VBI0` task removes a digitize buffer from the `next_digitize` queue and places it on the `digitizing` queue. Because each digitize buffer goes through this two-stage process, the `VBI0` task only performs `compress` operations on buffers that were used in a `digitize` operation occurring prior to the previous `VBI0` task.

Figure 2-23 shows the task declaration. The relative deadline is set to 15 ms. The resource list includes four global variables that this task shares with other tasks: the `next_digitize` queue, the `compress_source` queue, the `compress_sink` queue, and the pool of free compress buffers. The other global used by this task, the `digitizing` queue, is not used by any other task, and thus need not be included on the resource list. In order to pass the digitize buffer and the compress buffer to the next phase, the `CC` task, code is added to place these buffers on the `compress_source` and `compress_sink` queues.

```
task VBI0
deadline 15 ms
resources next_digitize,compress_source,compress_sink,compress_free
task body
   var db: digitize_buffer;
   var cb: compress_buffer;

   if length(digitizing) > 0 and available(compress_buffer) then
      db := remove_queue(digitizing);
      cb := allocate(compress_buffer);
      start_compress(db,cb);
      insert_queue(compress_source,db);
      insert_queue(compress_sink,cb);
   end if

   if length(next_digitizing) > 0 then
      db := remove_queue(next_digitizing);
      insert_queue(digitizing,db);
   end if
end task
```

Figure 2-23: Pseudo Code for VBI0 Task

```
task CC
deadline 8 ms
resources compress_sink, compress_free,
         transmit_video, video_transport
task body
   var cb: compress_buffer;

   if length(compress_sink) > 0 then
      cb := remove_queue(compress_sink);
      insert_queue(transmit_video,cb);
      video_transport := video_transport + 1;

      if video_transport > max_video_transport then
         video_transport := video_transport - 1;
         cb := remove_queue(transmit_video);
         free (cb);
      end if
   end if
end task
```

Figure 2-24: Pseudo Code for CC Task

The `CC` task is the application task corresponding to the last phase of the video process (see Figure 2-14). Figure 2-24 shows the task declaration. The relative deadline is set to 8 ms. The resource list includes four global variables that this task shares with other tasks: the pool of free compress buffers, the `compress_sink` queue, the `video_transmit` queue, and the `video_transport` counter. Note that as

discussed previously, the code to return the digitize buffer to the free pool is *not* included in this task.

The audio task is the application task corresponding to the single phase of the audio process (see Figure 2-16). Figure 2-25 shows the task declaration. The relative deadline is set to 15 ms. The resource list includes three global variables that this task shares with other tasks: the pool of free audio buffers, the transmit_audio queue and the audio_transport counter.

```
task audio
deadline 15 ms
resources transmit_audio, audio_transport, audio_free
task body
   var ab: audio_buffer;

   if available(audio_buffer) then
      ab := allocate(audio_buffer);
      audio_acquire(ab);
      insert_queue(transmit_audio,ab);
      audio_transport := audio_transport + 1;

      if audio_transport > max_audio_transport then
         audio_transport := audio_transport - 1;
         ab := remove_queue(transmit_audio);
         free(ab);
      end if
   end if
end task
```

Figure 2-25: Pseudo-Code for Audio Task

The initiate_send task is the application task corresponding to the first phase of the transport process (see Figure 2-17). Figure 2-26 shows the task declaration. The relative deadline is set to 20 ms. The resource list includes three global variables that this task shares with other tasks: the transmit_video queue, the transmit_audio queue, and the transmit_queue. The other global variable used by this task, the transmits_started counter, is not used by any other task, and thus need not be included on the resource list. In order to pass the packet descriptor to the next phase, implemented by the transmit_complete task, code is added to place the descriptor on the transmit_queue.

```
task initiate_send
deadline 20 ms
resources transmit_video, transmit_audio, transmit_queue
task body
   var
      d:  packet_descriptor;

   if eventcount(TC) < transmits_started then
      return;
   end if

   if length(transmit_video) > 0 then
      d.cb_count := 1;
      d.cb := remove_queue(transmit_video);
   else
      d.cb_count := 0;
   end if

   if length(transmit_audio) > 1 then
      d.ab_count := 2;
      d.ab1 := remove_queue(transmit_audio);
      d.ab2 := remove_queue(transmit_audio);
   else if length(transmit_audio > 0) then
      d.ab_count := 1;
      d.ab1 := remove_queue(transmit_audio);
   else
      d.ab_count := 0;
   end if

   transmits_started := transmits_started + 1;
   transmit(d);

   insert_queue(transmit_queue,d);
end body
```

Figure 2-26: Pseudo-Code for Initiate_Send Task

The `transmit_complete` task is the application task corresponding to the last phase of the transport process (see Figure 2-17). Figure 2-27 shows the task declaration. The relative deadline is set to 33 ms. The resource list includes five global variables that this task shares with other tasks: the `transmit_queue`, the `video_transport` counter, the `audio_transport` counter, the pool of free compress buffers and the pool of free audio buffers.

```
task transmit_complete
deadline 33 ms
resources transmit_queue,
         video_transport, compress_free,
         audio_transport, audio_free
task body
   var d: packet_descriptor;

   if length(transmit_queue) > 0 then
      d := remove_queue(transmit_queue);

      if d.cb_count > 0 then
         video_transport := video_transport - 1;
         free(d.cb);
      end if;

      if d.ab_count > 1 then
         audio_transport := audio_transport - 2;
         free(d.ab1);
         free(d.ab2);
      else if d.ab_count > 0 then
         audio_transport := audio_transport - 1;
         free(d.ab1);
      end if
   end if
end body
```

Figure 2-27: Pseudo-Code for Transmit_Complete Task

2.5.10 Miscellaneous Tasks and Interrupts

In addition to the interrupt handlers and tasks that form the application, there are several more interrupt handlers and tasks that execute during a run of the application. The `TIMER` interrupt handler runs in response to timer interrupts from the PS/2. The main function of the handler is to update the internal timekeeping data structures of YARTOS. In addition, it sends messages to two tasks that control user interactions with the application. Once every 9 executions (approximately 0.5 sec.), the `TIMER` handler sends a message to the `keyboard_check` task which polls for user input and responds to user commands. Once every 4 executions (approximately 2.0 sec), the `keyboard_check` task sends a message to the `screen_output` task that displays application status messages on the workstation monitor.

The ActionMedia adapters generate several other logical interrupts in addition to the VBI and CC logical interrupts described above. The handlers for these interrupts perform

some internal processing for the ActionMedia device driver[5]. One of these interrupts occurs at a regular rate once every 33 ms. and is handled by the DVI interrupt handler. The other interrupt occurs immediately after an `audio_acquire` operation is executed; this interrupt is handled by the DVI2 interrupt handler.

The network adapter generates several other logical interrupts in addition to the TC logical interrupt described above. Whenever the transmission of a new packet is initiated by a `transmit` call, the network adapter responds by generating two logical interrupts. In response to the first interrupt, the `NETWORK` handler performs limited processing on the packet. In response to the second interrupt, the `NETWORK` handler sends a message to an application task called `packet_transfer` that then copies the packet contents into memory on the network adapter. At this point, the network adapter transmits the data, although there may be some delay if the adapter cannot immediately access the physical network.

2.6 Summary and Discussion

In this chapter, I have described the workstation-based videoconferencing application that serves as the centerpiece of the dissertation. In particular, for the acquisition-side of the application, I have explained the implementation details that are needed for the performance analysis described in Chapters 3, 4 and 5.

I began with a high level description of the application. I then presented YARTOS, a real-time operating system kernel that supports a programming model in which interrupt handlers, operating system services, and application code execute to completion before well-defined deadlines. Next, I described the mechanics of acquiring, compressing, and transmitting audio and video frames. Finally, I presented pseudo-code descriptions of the set of YARTOS interrupt handlers and application tasks that comprise the acquisition-side of the application.

This chapter has concentrated on the acquisition-side of the application, but the display-side implementation has much in common with the acquisition-side. For example, the

---

[5]These interrupts are a good example of the operational nature of my understanding of the hardware interface to the ActionMedia adapters. I can determine when these interrupts occur, and I can verify that the handlers require very little execution time. However, I do not know what processing the handlers perform.

majority of audio and video processing on the display-side occurs in response to VBI interrupts. Audio frames are played by a task that executes in response to VBI interrupts. Video frames are decompressed by a task that executes in response to VBI0 interrupts and displayed by another task that executes in response to VBI1 interrupts.

In general, the implementation of the display-side can be thought of as the "inverse" of the acquisition-side. As on the acquisition-side, the design and implementation of the display-side can be described in terms of three high-level abstract processes: a transport process that receives frames from the network and puts them on queues, a video process that takes video frames from a transport queue, decompresses, and displays them, and an audio process that takes audio frames from a transport queue and plays them. As a result, the set of interrupt handlers and applications tasks on the display-side is similar to the set on the acquisition-side, except that data flows in the opposite directions. Thus, while the performance analysis described in Chapters 3, 4 and 5 is for the acquisition-side of the application, analysis of the display-side of the application would be quite similar.

Chapter III
Feasibility Analysis of YARTOS Task Systems

3.1 Introduction

The workstation-based conferencing application described in Chapter 2 is subject to a number of timing constraints. One constraint arises from properties of the ActionMedia hardware: to prevent a newly acquired digitized video frame from being overwritten, the application must provide a new digitize buffer to the ActionMedia video subsystem within a well-defined interval (see page 32). Another timing constraint arises from the application requirements: if frames are to be played with acceptable display latency, then the delay experienced by a frame at the acquisition workstation and at the display workstation must be bounded. These constraints are typical in *hard real-time* systems for which the correctness of a system depends on the system adhering to timing constraints.

One useful tool for designing, implementing, and analyzing hard real-time systems is the theory of deterministic scheduling and resource allocation as first developed by Liu and Layland [31]. Liu and Layland defined a formal model of real-time systems in which a real-time system consists of a set of tasks that must execute to completion prior to a deadline. A set of tasks is called *feasible* if there exists a scheduling discipline that always results in each task in the set executing to completion prior to its deadline. The authors propose two scheduling disciplines for their model; for each discipline they develop sufficient conditions for guaranteeing that a set of tasks scheduled under the discipline is feasible. I refer to these conditions as *feasibility conditions* and I refer to an algorithm for testing the conditions as a *feasibility test*.

Feasibility is an important concept because the knowledge that tasks in a system will always execute to completion prior to a deadline can be an extremely useful tool for analyzing a real-time system. However, we may often wish to determine if processing performed by a collection of tasks adheres to timing constraints that are not easily expressed in terms of a single deadline. For example, can we show that the time required to process a video frame is bounded given that processing of a video frame is a

combination of activities performed by several tasks? Simply knowing that each task completes prior to a deadline is not in general sufficient to determine such properties.

Another approach to the problem of demonstrating that real-time systems adhere to timing constraints uses a formal logic extended to account for timing behavior. Examples of this approach include Jahanian and Mok's Real-Time Logic (RTL) [22] and a method proposed by Shankar [47]. However, analysis of a large system with such a formal logic is often complex. One reason for this complexity is the need to reason about all possible orderings of events and task executions, which in turn are dependent on the particular scheduling discipline used in the system as well as details such as the time required to execute operations.

It is possible however to design a system so that its correctness is independent of the particular scheduling discipline and ordering of events; rather the correctness of a design will depend only on the fact that tasks will execute to completion prior to a specified deadline and will access data in accordance with specified mutual exclusion constraints. In such a case, the complexity of using a formal logic to verify the correctness of a system can be substantially reduced. In this work, I use a combination of scheduling theory and Real-Time Logic (RTL) to show that the delay experienced by video frames during processing on the acquisition-side of the application is bounded at 100 ms.

Bounding delay is the key to reducing or eliminating the delay jitter experienced by frames. As long as an upper bound on delay is known, delay jitter can be reduced or eliminated simply by buffering the frames to account for the difference between the actual delay experienced by the frame and the upper bound. Thus, by showing that the delay experienced by video frames during processing on the acquisition-side of the application is bounded, I will demonstrate that it is feasible to reduce or eliminate the delay jitter experienced by video frames on the acquisition-side.

While in principle an arbitrary bound on delay can be used to reduce or eliminate delay jitter, in practice such a bound should be reasonably tight. A loose bound would result in buffering a large amount of data, and more importantly would result in unnecessarily high display latency. While I do not formally derive lower bound on the delay experienced by video frames on the acquisition-side, I will argue in Chapter 5 that it must be at least 55 ms., even under assumptions that software operations take no time, and that work is always performed as soon as possible. Thus, the upper bound on delay of 100 ms. shown here is reasonably tight.

There are three steps in the analysis of the upper bound on delay. First, a feasibility test is used to demonstrate that each task in the system will execute to completion prior to its deadline and adhere to its mutual exclusion constraints. Next, an axiomatic specification of the system is developed (in RTL) in which the fact that tasks execute with these properties is included as an axiom. Finally, the bounded delay property is shown to be a theorem derivable from the axiomatic specification.

In this chapter, I define an abstract model of real-time systems that is implementable using the programming model of YARTOS. For this abstract model, I develop feasibility conditions that can be used to show that application tasks in a YARTOS application always execute to completion prior to a deadline and that access to resources by tasks is mutually exclusive. In Chapter 4, I apply these feasibility conditions to the workstation-based videoconferencing application. Finally, in Chapter 5, I develop the axiomatic specification of the application and the proof that the processing delay for video frames is bounded. In addition, I will argue that similar proofs can be developed to show that the delays experienced by audio frames on the acquisition-side, and by both audio and video frames on the display side are bounded.

The remainder of this chapter is organized as follows. Section 3.2 describes the abstract model of tasks that matches the execution model of YARTOS. Sections 3.3 and 3.4 develop two theorems about this model; Section 3.3 gives an upper bound on the processor time spent executing interrupt handlers and Section 3.4 shows that the scheduling discipline used by YARTOS enforces mutual exclusions constraints on task executions. Section 3.5 uses these results to derive complete feasibility conditions for the abstract model. Section 3.6 shows how these conditions can be used as the basis of a practical feasibility test.

3.2 System Model

As described in Section 2.4, the YARTOS programming model provides three primitives to an application developer: interrupt handlers, application tasks, and resources. Interrupt handlers are programs that execute in response to hardware interrupts. Application tasks are programs that execute in response to messages sent from interrupt handlers or other tasks. Resources are synchronization primitives that provide mutually exclusive access to tasks that share data. In this section, I define an abstract model of real-time systems that reflects this programming model.

I assume that real-time systems are systems in which software executes in response to the occurrence of *sporadic events*. A sporadic event is defined as a stimulus that is generated repeatedly with a lower bound on the interval between consecutive occurrences (*e.g.*, interrupts from a hardware timer). This lower bound is called the *minimum interarrival time* of the event. Specifically, if a sporadic event E has a minimum interarrival time of p, and t_i is defined as the time of the ith occurrence of E, then for all $i \geq 1$, $t_{i+1} \geq t_i + p$.

In my model, a real-time system is assumed to consist of a set of sequential programs called *tasks* that execute in response to sporadic events. Whenever an event occurs, the corresponding task is said to be *invoked*. An *invocation* of a task T is a copy of the sequential program of T which is created when the task is invoked. Each invocation of a task executes independently.

Tasks are divided into two distinct classes: *interrupt handlers* and *application tasks*. These classes differ in the rules governing a *correct* execution of a real-time system. For an execution to be considered correct, the scheduling of interrupt handler invocations and application task invocations must adhere to several constraints.

- An interrupt handler executes whenever one is available.
- Application task invocations complete execution prior to their deadlines.
- Resource usage by application tasks is mutually exclusive.

Formally, a *real-time task system* τ is defined as a set of m interrupt handlers $\{I_1, I_2, ..., I_m\}$, n application tasks $\{T_1, T_2, ..., T_n\}$, and r resources $\{R_1, R_2, ..., R_r\}$. An interrupt handler I is a pair (e, a) where e is the cost and a is the minimum interarrival time of the handler. The *cost* of an interrupt handler is defined as the maximum amount of processor time required to execute the handler to completion on a dedicated uniprocessor. The *minimum interarrival time* of an interrupt handler is defined as the minimum interarrival time of the event that invokes the handler. An application task T is a 4-tuple (c, U, d, p) where c is the cost of T, U is the set of resources used by T, d is the relative deadline of T, and p is the minimum interarrival time of T. The cost of an application task is defined as above. Each application task is said to *use* a subset (possibly empty) of the resources in τ. The *relative deadline* of an application task is defined as the length of the interval in which an invocation of the task must execute; if a task is invoked at time t, the invocation is assigned a deadline of $t + d$. Finally, the minimum interarrival time of an application task is defined as the minimum interarrival time of the event that invokes the application task.

A task system is *feasible* if, for an arbitrary sequence of interrupt handler and application task invocations, it is possible to schedule the task system correctly on a single processor.

The formal model is based on several additional assumptions. First, except for application tasks that share a resource, execution is assumed to be fully preemptive in the sense that a higher priority task is allowed to preempt a lower priority task at any time. Second, it is assumed that time is measured in discrete units. That is, interrupts and task invocations occur at *clock ticks* and parameters c, d, p, a, and e are expressed as integer multiples of the interval between successive clock ticks. Finally, it is assumed that application tasks and interrupt handlers are independent in the sense that the time at which a task is invoked is unrelated to any invocation of any other task (other than the previous invocation of the same task).

In the original model proposed by Liu and Layland [31], tasks are defined with respect to a number of constraints:

- Synchronous. Each task is invoked at time 0.
- Periodic. Each task is invoked periodically, *i.e.*, if a task has a minimum interarrival time of p, it is invoked every p time units.
- Deadline equal period. The relative deadline of a task is defined to be equal to the period of the task.
- Fully preemptive. A scheduling algorithm is allowed to preempt the execution of a task invocation at any time.
- Specific priority assignments. The theory assumes a specific assignment of priorities to task invocations.

Since Liu and Layland, a number of authors have defined and analyzed models of real-time tasks that relax one or more of these restrictions. The formal model defined in this chapter is a generalization of three of these models. In [4], Baruah, Mok, and Rosier developed feasibility conditions for a model consisting of sporadic tasks with arbitrary deadlines (*i.e.*, relaxing the synchronous, periodic, and deadline constraints). In [25], Jeffay derived feasibility conditions for a model in which sporadic tasks share resources (*i.e.*, relaxing the fully preemptive constraint). Finally, in [26], Jeffay and Stone derived feasibility conditions for a model that included both periodic interrupt handlers and periodic application tasks (*i.e.*, relaxing the algorithm-defined priority constraint). The model I propose here relaxes all of these constraints. Because this model combines several

properties of these three models, a number of the proofs in this chapter have been adapted from the proofs in these three papers.

3.3 The Effect of Interrupt Handlers

As the first step in deriving the feasibility conditions for real-time task sets, I examine the effect of interrupt handler execution on the time required to complete invocations of application tasks. Because interrupt handler invocations execute with priority strictly greater than application task invocations, the effect of executing interrupt handlers is to reduce the amount of time available to execute invocations of application tasks. To quantify the time spent executing interrupt handler invocations, consider a task system τ that includes m interrupt handlers $(e_1, a_1) \ldots (e_m, a_m)$. Let $f(l)$ be a function from the non-negative integers to the non-negative integers defined by the following recurrence relation:

$$f(0) = 0,$$

$$\forall l > 0, f(l) = \begin{cases} f(l-1) & \text{if } f(l-1) = \sum_{i=1}^{m} \left\lceil \frac{l}{a_i} \right\rceil e_i \\ f(l-1)+1 & \text{if } f(l-1) < \sum_{i=1}^{m} \left\lceil \frac{l}{a_i} \right\rceil e_i \end{cases}$$

As shown in the following theorem, $f(l)$ is an upper bound on the amount of time spent executing interrupt handler invocations in an arbitrary interval of length l of an execution of τ. The definition can be interpreted by considering the worst-case execution of τ, in which every interrupt handler is invoked at time 0 and periodically thereafter. In this case, at any time l, $f(l)$ is exactly the time that was spent executing interrupt handlers in the interval $[0, l]$. The two cases in the definition correspond to whether or not all the interrupt handlers that were requested in the interval $[0, l-1]$ completed prior to $l-1$; if so, then $f(l) = f(l-1)+1$ since an interrupt handler was executed in the interval $[l-1, l]$.

One additional note about the definition of $f(l)$ is useful. For all $l > 0$,

$$f(l-1) \leq f(l) \tag{3.1}$$

Theorem 3.1: Let τ be a task system with m interrupt handlers $(e_1, a_1) \ldots (e_m, a_m)$. For all $a \geq 0$, $b \geq a$, let $g(a,b)$ be the amount of processor time consumed by interrupt handler invocations in the interval $[a, b]$ during an arbitrary execution of τ. For all t and l, $t \geq 0$, $l \geq 0$, $f(l)$ is an upper bound on $g(t, t+l)$ (*i.e.*, $g(t, t+l) \leq f(l)$).

Proof (adapted from a proof given in [26]): By contradiction.

Suppose $f(l)$ is not an upper bound on $g(t, t+l)$ for all $t \geq 0$ and $l \geq 0$. Then there exists some x and k such that

$$f(k) < g(x, x+k) \tag{3.2}$$

Choose the smallest x for which (3.2) holds, and for that x, choose the smallest k. This choice has several consequences:

1. $k > 0$. Since the amount of processor time consumed by interrupt handler invocations in an interval of length 0 is necessarily 0, for all x, $g(x, x) = 0$. Since by definition $f(0) = 0$, $g(x, x) \leq f(0)$.

2. For $t < x$, $g(t, t+l) \leq f(l)$. Since x is chosen to be the smallest value of t for which $f(l)$ is not an upper bound on $g(t, t+l)$, $f(l)$ is an upper bound for all intervals starting prior to x.

3. For $l < k$, $g(x, x+l) \leq f(l)$. Since k is chosen to be the smallest value for which $f(l)$ is not an upper bound on $g(x, x+l)$, $f(l)$ is an upper bound for all intervals starting at x with length less than k.

The assumption that $f(k)$ is not a bound on the amount of processor time consumed by interrupt handler invocations in the interval $[x,\ x+k]$ (*i.e.*, equation 3.2) can be combined with equation (3.1) and fact 3 for $l = k-1$ to produce:

$$\begin{aligned} g(x, x+k-1) &\leq f(k-1) \\ &\leq f(k) \\ &< g(x, x+k) \end{aligned} \tag{3.3}$$

The interpretation of equation (3.3) is that the amount of time spent executing interrupt handler invocations in the interval $[x,\ x+k-1]$ is strictly less than the time spent executing interrupt handler invocations in the interval $[x,\ x+k]$. As a result, the processor must have executed an invocation of an interrupt handler in the unit interval $[x+k-1,\ x+k]$. Thus

$$g(x+k-1, x+k) = 1 \tag{3.4}$$

Hence

$$
\begin{aligned}
g(x, x+k) &= g(x, x+k-1) + g(x+k-1, x+k) \\
&= g(x, x+k-1) + 1 \\
&\leq f(k-1) + 1
\end{aligned} \tag{3.5}
$$

It follows from (3.2) and (3.5) that $f(k) < f(k-1)+1$. Since for all $l > 0$, $f(l-1) \leq f(l)$, it then follows that $f(k) = f(k-1)$. Thus by the definition of *f*(*l*),

$$
f(k) = \sum_{i=1}^{m} \lceil k/a_i \rceil e_i \tag{3.6}
$$

Now, there are two cases that must be considered, depending on the value of *x*. If $x > 0$, then by fact 2 for $t = x-1$, $g(x-1, x+k-1) \leq f(k)$. Combining this with assumption (3.2),

$$
\begin{aligned}
g(x-1, x+k-1) &\leq f(k) \\
&< g(x, x+k)
\end{aligned}
$$

Thus

$$
\begin{aligned}
g(x-1, x) + g(x, x+k-1) &< g(x, x+k-1) + g(x+k-1, x+k) \\
g(x-1, x) &< g(x+k-1, x+k)
\end{aligned}
$$

It follows from this and equation (3.4) that $g(x-1, x) < 1$, and thus $g(x-1, x) = 0$. Therefore, during the unit interval $[x-1,\ x]$, the processor did not execute an interrupt handler invocation. Thus, all invocations of interrupt handlers occurring prior to *x* must have completed execution before *x*.

On the other hand, if $x = 0$, then by definition there were no invocations of interrupt handlers that occurred prior to *x*. Thus, independent of the value of *x*, the only interrupt handler invocations that executed in the interval $[x,\ x+k]$ were those invoked at or after *x*.

Since an invocation of an interrupt handler can only execute in an interval [*a*, *b*] if it is invoked at or before $b-1$, the only interrupt handler invocations that executed in the interval $[x,\ x+k]$ were those invoked in the interval $[x,\ x+k-1]$. An upper bound on the number of invocations of an interrupt handler (*e*, *a*) in this interval is given by $\lceil k/a \rceil$ and thus an upper bound on the total processing requirement of interrupt handler invocations

occurring in the interval is given by $\sum_{i=1}^{m} \lceil k/a_i \rceil e_i$. Thus, $g(x, x+k) \le \sum_{i=1}^{m} \lceil k/a_i \rceil e_i$. It then follows from (3.6) that

$$g(x, x+k) \le \sum_{i=1}^{m} \lceil k/a_i \rceil e_i = f(k) \qquad (3.7)$$

which contradicts the assumption that $f(k) < g(x, x+k)$. □

Theorem 3.1 shows that in any interval of length L, at most $f(L)$ units of processor time are expended executing interrupt handlers. Thus it is the case that in any interval of length L, at least $L - f(L)$ units of processor time are available to execute application tasks. This fact will be used in Section 3.5 as part of the derivation of feasibility conditions.

3.4 EDF/DDM Scheduling Discipline

Next, I describe the scheduling discipline used in YARTOS to schedule application tasks and show that the use of this discipline is sufficient to enforce the property that tasks that share resources access those resources in a mutually exclusive manner. This scheduling discipline is called *Earliest Deadline First with Dynamic Deadline Modification* (EDF/DDM), a (trivial) variant of the scheduling discipline proposed by Jeffay in [25].

The EDF/DDM scheduling discipline operates as follows. At any time there is an application task eligible for execution, the task invocation with the nearest *contending deadline* is executed. The contending deadline is initially defined as the deadline of the invocation. However, once the invocation has begun executing, its contending deadline is modified (as explained below). In the case tie contending deadlines, task invocations that are preempted are given precedence; otherwise the choice is arbitrary.

Consider a task system with m interrupt handlers, n application tasks, and r resources, defined as above. For task T_i, let D_i represent the smallest relative deadline of any application task with which it shares a resource. That is,

$$D_i = \min\left\{ d_j \middle| 1 \le j \le n \wedge U_i \cap U_j \ne \varnothing \right\}$$

Let t_i be a time at which T_i is invoked. The deadline of the invocation of T_i occurring at t_i is defined as $t_i + d_i$. Thus, the initial value of the contending deadline of this invocation is also $t_i + d_i$. When the invocation begins execution, say at time $t_s \ge t_i$, its contending

deadline is modified to the earlier of $t_s + D_i + 1$ or the original deadline. That is, once a task invocation begins execution, its contending deadline is given by

$$\min\left(t_s + D_i + 1, t_i + d_i\right)$$

Because $D_i \leq d_j$ for all tasks T_j with which T_i shares a resource, this contending deadline is guaranteed to be nearer than the initial deadline of any invocation of T_j that occurs after the invocation of T_i begins execution. Thus, tasks which share resources and are scheduled under the EDF/DDM scheduling discipline are guaranteed not to preempt one another. Thus, they access resources in a mutually exclusive manner. The following theorem demonstrates this principle.

Theorem 3.2: Invocations of application tasks in a task set τ scheduled with the EDF/DDM scheduling discipline access resources in a mutually exclusive manner.

Proof (adapted from a proof given in [25]): It suffices to show that under the EDF/DDM scheduling discipline, an invocation of a task that requires resource R_k cannot begin execution if an invocation of another task that requires R_k has begun but not yet completed execution.

Let T_i be a task that requires R_k, let t_i be a point in time when T_i is invoked, and let t_s be the point in time at which that invocation of T_i first commences execution. Let T_j be another task that requests R_k. The proof shows that an invocation of T_j cannot begin execution if the invocation of T_i has begun but not yet completed execution. This is shown by contradiction.

Let $t > t_s$ be a point in time when the invocation of T_i has begun but not yet completed execution and assume that an invocation of T_j begins execution at t. Let t_j be the time at which this invocation of T_j occurred.

Because it started executing prior to time t, the invocation T_i has a contending deadline of $\min\left(t_s + D_i + 1, t_i + d_i\right)$ at time t. Since T_j has not yet begun execution at time t, it contends for the processor at time t with its initial deadline of $t_j + d_j$. Because T_j is chosen for execution, it must be the case that this deadline is nearer than the contending deadline of task T_i. Thus,

$$t_j + d_j < \min\left(t_s + D_i + 1, t_i + d_i\right) \tag{3.8}$$

Since T_i began execution at t_s, either or both of the following facts are true

- The invocation of T_j occurred after the invocation of T_i began execution. Thus, $t_j > t_s \geq t_i$.
- The invocation of T_i had a nearer initial deadline than the invocation of T_j. Thus, $t_j + d_j \geq t_i + d_i$.

If neither of these facts were true, then T_j would have been executed at t_s since it would have been available for execution and would have had a nearer deadline than the invocation of T_i.

Because tasks T_i and T_j share a resource, it is necessarily the case that $D_i \leq d_j$. Thus, if the invocation of T_j occurred after the invocation of T_i began execution, then

$$\begin{aligned} t_j + d_j &\geq t_j + D_i \\ &> t_s + D_i \\ &\geq t_s + D_i + 1 \\ &\geq \min\left(t_s + D_i + 1, t_i + d_i\right) \end{aligned}$$

which contradicts (3.8).

If the invocation of T_i had a nearer initial deadline than the invocation of T_j, then

$$\begin{aligned} t_j + d_j &\geq t_i + d_i \\ &\geq \min\left(t_s + D_i + 1, t_i + d_i\right) \end{aligned}$$

which also contradicts (3.8). □

This theorem has shown that any task system scheduled with the EDF/DDM discipline will enforce mutually exclusive access to resources. This fact will be used in the next section in the derivation of feasibility conditions for task sets.

3.5 Feasibility Conditions

I am now ready to develop feasibility conditions for the model described in Section 3.2. To begin, it is useful to quantify the maximum time required to complete execution of all task invocations that occur in an interval. Consider an interval $[t,\ t+L]$ and a task T_i. How many invocations of T_i can occur at or after t with a deadline at or before $t+L$? The maximum number of invocations meeting this criteria will occur when an invocation occurs at t and subsequent invocations occur periodically (*i.e.*, tasks are invoked at $t+kp_i$ and have deadlines at $t+kp_i+d_i$ for $k \geq 0$). If $L < d_i$, then the invocation occurring at t has a deadline after $t+L$, so there are no invocations meeting the criteria. If $L \geq d_i$, then the deadline of the invocation occurring at t is in the interval. Furthermore, the number of additional invocations with deadlines at or before $t+L$ is equal to $\lfloor (L-d_i)/p_i \rfloor$. Thus for each task T_i, an upper bound on the number of invocations occurring at or after t with a deadline at or before $t+L$ is

$$\delta_i(L) = \begin{cases} 0 & \text{if } L < d_i \\ 1 + \left\lfloor \dfrac{L-d_i}{p_i} \right\rfloor & \text{if } L \geq d_i \end{cases}$$

Therefore, for each task T_i, an upper bound on the number of invocations occurring at or after t_0 with a deadline at or before t_d is $\delta_i(t_d - t_0)$. Thus, an upper bound on the processing requirement of these invocations of T_i is $\delta_i(t_d - t_0) \cdot c_i$.

For a task set to be feasible, two conditions must hold. First, at any given time, the amount of time that has been available for executing application tasks up to that point must be at least as great as the total processing requirement of all application tasks up to that point. In the interval [0, L], the amount of time available for executing application tasks is at least L minus the maximum time that could have been spent executing interrupt handlers, or $L - f(L)$. An upper bound on the processing requirement of application tasks invocations in the interval [0, L] is the sum of the requirements for each task. Thus, the first feasibility condition is

$$\forall L, L \geq 0, \quad L - f(L) \geq \sum_{i=1}^{n} \delta_i(L) \cdot c_i$$

The second condition that must hold for a task set to be feasible addresses the effect of the preemption constraint introduced by shared resources. For each task, the following feasibility condition must hold:

$$\forall L, D_i < L < d_i$$
$$L - f(L) \geq c_i + \sum_{j=1}^{n} \delta_j (L-1) \cdot c_j$$

This condition can be interpreted by considering a particular worst-case sequence of task and interrupt handler invocations. Assume that a task T_i is invoked at some time $t_0 - 1$. Then, at time t_0, every other application task is invoked and each task is invoked periodically thereafter. The condition shows that at all times in the interval $[t_0 + D_i$, $t_0 + d_i]$, there is enough time available to meet the processing requirement of all task invocations with deadlines in the interval. The following theorem shows that these two conditions are sufficient to show that a task set is feasible.

Theorem 3.3: Let τ be a task system with m interrupt handlers $\{(e_1, a_1), ..., (e_m, a_m)\}$, n application tasks $\{(c_1, U_1, d_1, p_1), ..., (c_n, U_n, d_n, p_n)\}$ and r resources $\{R_1, R_2, ... R_r\}$. τ will be feasible if the following two conditions hold.

1) $\forall L, L \geq 0,$
$$L - f(L) \geq \sum_{i=1}^{n} \delta_i (L) \cdot c_i$$

2) $\forall i, 1 \leq i \leq n, \forall L, D_i < L < d_i$
$$L - f(L) \geq c_i + \sum_{j=1}^{n} \delta_j (L-1) \cdot c_j$$

Proof: To prove the theorem, it must be shown that when Conditions 1 and 2 hold for a task set τ, the EDF/DDM scheduling discipline will succeed in scheduling the tasks in τ so that access to resources is mutually exclusive and so that tasks always execute to completion prior to their deadline. Theorem 3.2 has already shown that the EDF/DDM scheduling discipline maintains the mutual exclusion constraints on access to resources, independent of Conditions 1 and 2. Thus, it remains to show that tasks meet their deadlines. This is shown by contradiction.

Assume that Conditions 1 and 2 hold, and yet a task invocation fails to execute to completion prior to its deadline when the task set is scheduled under the EDF/DDM scheduling discipline. Let t_d be the earliest point in time at which a task invocation misses its deadline. Let t_0 be the latest of:

- 0
- The end of the last period in which the processor was idle prior to t_d.
- The last time prior to t_d a task invocation with both a deadline and a contending deadline after t_d stopped execution (defined as t_d if such a task was still executing at t_d).
- The last time prior to t_d a task invocation with a deadline after t_d and a contending deadline at or before t_d started execution.

As a result of this definition of t_0, several facts hold:

1. The processor executed continuously in the interval $[t_0, t_d]$. If this was not the case, then there would be some time after t_0 and prior to t_d during which the processor was idle. This is prohibited by the choice of t_0.

2. By Theorem 3.1, it is the case that at most $f(t_d - t_0)$ units of processor time were spent executing interrupt handlers in $[t_0, t_d]$. Therefore, because the processor executed continuously, at least $(t_d - t_0) - f(t_d - t_0)$ units of processor time were spent executing invocations of application tasks in the interval $[t_0, t_d]$.

3. Every task invocation occurring prior to t_0 with a deadline at or before t_d completed execution prior to t_0. Thus, any task invocation that misses a deadline at t_d must have been invoked at or after t_0. To show this fact, four cases must be considered corresponding to the four restrictions on the choice of t_0 defined above. First, if $t_0 = 0$, then by definition there were no task invocations that occurred prior to t_0. Second, if the processor was idle immediately prior to t_0, then there were no task invocations that occurred prior to t_0 that had not already completed execution by t_0. Third, if the processor stopped executing a task with a contending deadline after t_d at time t_0, then at time $t_0 - 1$ there were no outstanding task invocations with a deadline prior to t_d. Finally, if a task with a deadline after t_d started executing at time t_0, then there were no task invocations with deadlines at or before t_d that occurred prior to t_0 that had not already completed execution by t_0.

4. At most one task invocation with a deadline after t_d executed in the interval $[t_0, t_d]$. Furthermore, this task invocation began execution at t_0 and had a contending deadline at or before t_d. This fact is a consequence of the fact that t_0 was chosen to be greater than or equal to the last time a task invocation with a deadline after t_d began execution.

5. The only other task invocations executed by the processor in the interval $[t_0, t_d]$ were those that were invoked at or after t_0 with deadlines at or before t_d. This fact results from facts 3 and 4.

Next, I use these facts to derive an upper bound on the total processing required to complete execution of all task invocations that can execute in the interval $[t_0, t_d]$. I then show that Conditions 1 and 2 are sufficient to guarantee that this requirement is met. Since all task invocations with deadlines at or before t_d either complete execution prior to t_0 or are eligible to execute at some point in the interval $[t_0, t_d]$, this will show that every task invocation with a deadline at or before t_d will complete execution at or before t_d. This will contradict the assumption that a task misses a deadline at t_d.

There are two cases to be considered depending on whether or not there is a task invocation with a deadline after t_d which executes in the interval $[t_0, t_d]$.

<u>Case 1</u>: Assume that no task invocation with a deadline after t_d executes in $[t_0, t_d]$. Then by facts 4 and 5, only task invocations that occurred at or after t_0 with deadlines at or before t_d were executed by the processor in the interval $[t_0, t_d]$. An upper bound on the total processing required to complete execution of all task invocations occurring at or after t_0 with deadlines at or before t_d is

$$\sum_{i=1}^{n} \delta_i(t_d - t_0) \cdot c_i$$

By fact 2, at least $(t_d - t_0) - f(t_d - t_0)$ units of processor time were spent executing invocations of application tasks in the interval $[t_0, t_d]$ and by Condition 1, it is the case that

$$(t_d - t_0) - f(t_d - t_0) \geq \sum_{i=1}^{n} \delta_i(t_d - t_0) \cdot c_i$$

Thus, it is the case that, in the interval $[t_0, t_d]$, the time spent executing invocations of application tasks invoked at or after t_0 with deadlines at or before t_d was at least as great as the total processing requirement of these task invocations. Thus, every task invoked at or after t_0 with a deadline at or before t_d must have completed execution at or before t_d.

Because by fact 1 any task invocation that misses a deadline at t_d must have been invoked at or after t_0, and because t_d was chosen to be the earliest time at which a task invocation misses a deadline, this implies that no task invocation misses a deadline at or before t_d, which contradicts the assumption that a task missed a deadline at t_d.

Case 2: Assume that an invocation of task T_i with a deadline after t_d executes in $[t_0, t_d]$ and further assume that this invocation occurred at s_i. By fact 4, this invocation began executing at t_0, so $s_i \le t_0$. Thus, it is the case that

$$\begin{aligned} s_i + d_i &> t_d \\ d_i &> t_d - s_i \\ d_i &> t_d - t_0 \end{aligned} \tag{3.9}$$

In addition, by fact 4, this invocation of task T_i must have a contending deadline at or before t_d. Thus, $\min\left(t_0 + D_i + 1, s_i + d_i\right) \le t_d$. Because it has been assumed that the deadline the task invocation is after t_d, it is the case that $s_i + d_i > t_d$. Thus

$$\begin{aligned} \min\left(t_0 + D_i + 1, s_i + d_i\right) &\le t_d \\ t_0 + D_i + 1 &\le t_d \\ D_i &\le t_d - t_0 - 1 \\ D_i &< t_d - t_0 \end{aligned} \tag{3.10}$$

Combining (3.9) and (3.10) gives $D_i < t_d - t_0 < d_i$.

By fact 4, the invocation of task T_i with a deadline after t_d that executes in $[t_0, t_d]$ began executing at t_0. Furthermore, because this invocation began executing at t_0 with a deadline after t_d, no other task could have been invoked at t_0 with a deadline at or before t_d (any such invocation would have been chosen for execution at t_0). Thus, in addition to the invocation of T_i, only task invocations that occurred at or after $t_0 + 1$ with deadlines at or before t_d were executed by the processor in the interval $[t_0, t_d]$. An upper bound on the total processing required to complete execution of these task invocations is thus

$$c_i + \sum_{j=1}^{n} \delta_j\left(t_d - \left(t_0 + 1\right)\right) \cdot c_j$$

By fact 2, at least $(t_d - t_0) - f(t_d - t_0)$ units of processor time were spent executing invocations of application tasks in the interval $[t_0, t_d]$ and because $D_i < t_d - t_0 < d_i$, Condition 2 implies

$$(t_d - t_0) - f(t_d - t_0) \geq c_i + \sum_{j=1}^{n} \delta_j\big(t_d - (t_0 + 1)\big) \cdot c_j$$

Thus, it is the case that, in the interval $[t_0, t_d]$, the time spent executing either invocations of application tasks invoked at or after t_0 with deadlines at or before t_d or the invocation of task T_i that occurred at s_i was at least as great as the total processing requirement of these task invocations. Thus, the invocation of T_i that occurred at s_i and every task invocation occurring at or after t_0 with a deadline at or before t_d must have completed execution at or before t_d. Because by fact 1 any task invocation that misses a deadline at t_d must have been invoked at or after t_0, and because t_d was chosen to be the earliest time at which a task invocation misses a deadline, this implies that no task invocation misses a deadline at or before t_d, which contradicts the assumption.

Thus I have shown that in all cases, if Conditions 1 and 2 hold for a task set τ, then the EDF/DDM scheduling discipline will succeed in scheduling the tasks in τ so that access to resources is mutually exclusive and so that tasks always execute to completion prior to their deadlines. □

3.6 Feasibility Test

While Theorem 3.3 gives sufficient conditions for the feasibility of a task set, the requirement that Condition 1 hold for all $L \geq 0$ implies that it cannot be used directly as the basis of a practical feasibility test. However, for most task systems, it is possible to bound the values of L at which Condition 1 must be evaluated.

The *achievable processor utilization* (or utilization) of a task system is defined as an upper bound on the fraction of processor time that is required by the tasks over an arbitrarily long interval. The achievable process utilization of a task system τ is precisely expressed as:

$$\Psi_\tau = \sum_{i=1}^{n} \frac{c_i}{p_i} + \sum_{i=1}^{m} \frac{e_i}{a_i}$$

Theorem 3.5 shows that for task systems with achievable processor utilization strictly less than one, the feasibility conditions need only be applied to a bounded set values in order to guarantee that the task system is feasible. As a result, Theorem 3.5 can be used as the basis of a practical feasibility test.

Before presenting Theorem 3.5, I first prove a lemma needed for the proof. The intuition behind this lemma is simple: the function $\delta_i(t)$ only changes at values of t which are multiples of the minimum interarrival time of some application task.

Lemma 3.4: Let $Q = \{kp_i + d_i | k \geq 0 \wedge 1 \leq i \leq n\} \cup \{0\}$. Let t and t' be any two elements of Q such that $t < t'$ and there does not exist an $r \in Q$, $t < r < t'$. Let ε be an integer such that $0 \leq \varepsilon < t' - t$. For all i, $1 \leq i \leq n$, $\delta_i(t) = \delta_i(t + \varepsilon)$.

Proof: There are two cases to be considered.

Case 1: Assume that $t < d_i$. By the choice of t', it is the case that $t' \leq d_i$ and therefore that $t + \varepsilon < d_i$. Thus, $\delta_i(t + \varepsilon) = \delta_i(t) = 0$.

Case 2: Assume that $t \geq d_i$. Let k be the largest integer such that $t \geq kp_i + d_i$. Recall that $\delta_i(t)$ is defined as

$$\delta_i(t) = \begin{cases} 0 & \text{if } t < d_i \\ 1 + \left\lfloor \frac{t - d_i}{p_i} \right\rfloor & \text{if } t \geq d_i \end{cases}$$

Thus

$$\delta_i(t) = 1 + \left\lfloor \frac{t - d_i}{p_i} \right\rfloor = k + 1$$

By the choice of t', it is the case that $t' \leq (k+1)p_i + d_i$. Thus, by the choice of ε, it is the case that $t + \varepsilon < (k+1)p_i + d_i$. Thus

$$\begin{aligned}\delta_i(t+\varepsilon) &= 1+\left\lfloor \frac{t+\varepsilon-d_i}{p_i} \right\rfloor \\ &\geq 1+\left\lfloor \frac{t-d_i}{p_i} \right\rfloor \\ &\geq 1+\left\lfloor \frac{kp_i+d_i-d_i}{p_i} \right\rfloor \\ &\geq k+1\end{aligned}$$

and,

$$\begin{aligned}\delta_i(t+\varepsilon) &= 1+\left\lfloor \frac{t+\varepsilon-d_i}{p_i} \right\rfloor \\ &\leq 1+\frac{t+\varepsilon-d_i}{p_i} \\ &< 1+\frac{(k+1)p_i+d_i-d_i}{p_i} \\ &< k+2\end{aligned}$$

Thus, $\delta_i(t+\varepsilon)=\delta_i(t)=k+1$. In either case, $\delta_i(t+\varepsilon)=\delta_i(t)$. This proves the lemma. □

I am now ready to prove Theorem 3.5. This theorem is similar to Theorem 3.3, but restricts the set of points at which the feasibility conditions must be tested. In particular, it shows that if Condition 1 holds at a set of points defined by the multiples of the minimum interarrival times of each task and bounded by a value B_τ, then Condition 1 must hold at all L.

Theorem 3.5: Let τ be a task system with m interrupt handlers $\{(e_1, a_1), ..., (e_m, a_m)\}$, n application tasks $\{(c_1, U_1, d_1, p_1), ..., (c_n, U_n, d_n, p_n)\}$ and r resources $\{R_1, R_2, ... R_r\}$ defined such that $\Psi_\tau < 1$. Let

$$B_\tau = \frac{\sum_{i=1}^{m} e_i + \sum_{i=1}^{n} c_i}{1-\Psi_\tau}$$

and let $P=\left\{kp_i+d_i \middle| kp_i+d_i \leq B_\tau \wedge k \geq 0 \wedge 1 \leq i \leq n\right\} \cup \{0\}$. τ will be feasible if the following two conditions hold.

1) $\forall L, L \in P$

$$L - f(L) \geq \sum_{i=1}^{n} \delta_i(L) \cdot c_i$$

2) $\forall i, 1 \leq i \leq n, \forall L, D_i < L < d_i$

$$L - f(L) \geq c_i + \sum_{j=1}^{n} \delta_j(L-1) \cdot c_j$$

Proof: To prove the theorem, it is sufficient to show that the two conditions of this theorem imply the two conditions of Theorem 3.3. Condition 2 is identical to condition 2 in Theorem 3.3. Thus, it remains to show that Condition 1 of this theorem implies Condition 1 of Theorem 3.3.

Assume Condition 1 holds for a task set τ. That is, $\forall L, L \in P$

$$L - f(L) \geq \sum_{i=1}^{n} \delta_i(L) \cdot c_i \qquad (3.11)$$

Equation (3.11) is identical to the equation given in Condition 1 of Theorem 3.3. The difference is in the range of *L* at which the condition must be checked. I show that Condition 1 of this theorem implies Condition 1 of Theorem 3.3 by showing that when equation (3.11) holds for all $L \in P$, it must hold for all *L* greater than or equal to 0.

There are two parts to the proof. First, I show that if (3.11) holds for certain values of *L*, then it holds for every value of *L*. Second, I show that if (3.11) holds for all $L < B_\tau$, then it holds for all *L*.

Let $Q = \{kp_i + d_i | k \geq 0 \wedge 1 \leq i \leq n\} \cup \{0\}$. Choose $t, t' \in Q$, $t < t'$ such that there does not exist an $r \in Q$, $t < r < t'$. Let ε be an integer such that $0 \leq \varepsilon < t' - t$. By Lemma 3.4, it is the case that for all *i*, $\delta_i(t) = \delta_i(t + \varepsilon)$. Moreover, since at most ε time units can be spent executing interrupt handlers in the interval $[t, t + \varepsilon]$, it is the case that $f(t + \varepsilon) \leq f(t) + \varepsilon$. If (3.11) is satisfied at for $L = t$, then

$$t - f(t) \geq \sum_{i=1}^{n} \delta_i(t) \cdot c_i$$

$$t - f(t) \geq \sum_{i=1}^{n} \delta_i(t+\varepsilon) \cdot c_i$$

$$t + \varepsilon - \left(f(t) + \varepsilon\right) \geq \sum_{i=1}^{n} \delta_i(t+\varepsilon) \cdot c_i$$

$$t + \varepsilon - f(t+\varepsilon) \geq \sum_{i=1}^{n} \delta_i(t+\varepsilon) \cdot c_i$$

Therefore, if (3.11) holds for all elements of Q, it holds for all L.

Next, consider the function

$$g(L) = \sum_{i=1}^{n} \delta_i(L) \cdot c_i + f(L) - L$$

Equation (3.11) can be restated as $g(L) \leq 0$.

Let $h(L) = \left(U_\tau - 1\right) \cdot L + \sum_{j=1}^{m} e_j + \sum_{j=1}^{n} c_j$.

Noting that for all $L \geq 0$, $\delta_i(L) \leq 1 + \lfloor L/p_i \rfloor$ and $f(L) \leq \sum_{i=1}^{m} \lceil L/a_i \rceil e_i$, it is the case that

$$\begin{aligned} g(L) &\leq \sum_{j=1}^{n} \left(1 + \left\lfloor \frac{L}{p_j} \right\rfloor\right) c_j + \sum_{j=1}^{m} \left\lceil \frac{L}{a_j} \right\rceil e_j - L \\ &\leq \sum_{j=1}^{n} \left(1 + \frac{L}{p_j}\right) c_j + \sum_{j=1}^{m} \left(\frac{L}{a_j} + 1\right) e_j - L \\ &\leq \sum_{j=1}^{n} c_j + L \sum_{j=1}^{n} \frac{c_j}{p_j} + L \sum_{j=1}^{m} \frac{e_j}{a_j} + \sum_{j=1}^{m} e_j - L \\ &\leq \left(U_\tau - 1\right) \cdot L + \sum_{j=1}^{m} e_j + \sum_{j=1}^{n} c_j \\ &\leq h(L) \end{aligned}$$

Thus $h(L)$ bounds $g(L)$ from above. $h(L)$ is a linear function in L with slope $U(\tau) - 1$ and an L-intercept at the point

$$L = B_\tau = \frac{\sum_{i=1}^{m} e_i + \sum_{i=1}^{n} c_i}{1 - \Psi_\tau}$$

Since $\Psi_\tau < 1$, $h(L)$ has negative slope and thus for all $L > B_\tau$, $g(L) \le h(L) \le 0$. Hence if (3.11) holds for all $L \le B_\tau$, then it holds for all L.

The set P is the intersection of the set Q and the set of all $L \le B_\tau$. Thus, if equation (3.11) holds $\forall L, L \in P$, then it holds for all L greater than or equal to 0. This proves the theorem. □

3.7 Summary

In this chapter, I have defined an abstract model of real-time systems that is implementable using the programming model of YARTOS. For this abstract model, I have developed a practical feasibility test that can be used to show that, when scheduled according to a variant of the Earliest-Deadline First scheduling discipline, application tasks in a YARTOS application always execute to completion prior to a deadline and that access to resources by tasks is mutually exclusive. In the next chapter, I will apply these feasibility conditions to a specification of the workstation-based videoconferencing application.

Chapter IV
Feasibility Analysis of the Acquisition-Side

4.1 Introduction

In Chapter 3, I derived a feasibility test for an abstract model of real-time systems that matched the programming model of YARTOS. In this chapter, I use this feasibility test to show that the application tasks comprising the acquisition-side of the workstation-based video conferencing application described in Chapter 2 always execute to completion prior to their deadlines and that the tasks that share resources adhere to the required mutual exclusion constraints. In Chapter 5, the properties shown in this chapter will be included as axioms in an axiomatic specification of the application from which I derive the fact that the delays experienced by video frames on the acquisition-side are bounded.

As described in Chapter 2, the video conferencing application consists of a set of interrupt handlers, a set of application tasks, and a set of resources that execute on top of the YARTOS kernel. To use the feasibility test, I must represent these as a task system τ as defined in Section 3.2. This requires that I provide values for each of the parameters that characterize interrupt handlers and tasks in the formal model:

- the maximum execution cost of each interrupt handler and application task.
- the minimum interarrival time for each interrupt handler and application task.
- the relative deadline of each application task.
- the set of resources used by each application task.

In this chapter, emphasis is placed on determining minimum interarrival times for each interrupt handler and application task. In Section 4.2, I begin with a discussion of the hardware interrupts generated during execution and the difficulties involved in determining the minimum separation between occurrences of each interrupt and thus the minimum interarrival time of the interrupt handlers. In Section 4.3, I develop a formal method that addresses these difficulties. In Section 4.4, I discuss the techniques used to determine the

minimum interarrival time of application tasks. Finally, in Section 4.5, I put it all together, present the description of the application in terms of the formal model, and present the results of the feasibility test.

In the formal model presented in Chapter 3, time is discrete. Thus, an atomic time unit must be chosen to specify the execution costs, minimum interarrival times, etc., needed to represent the application in terms of the formal model. Throughout the analysis in this chapter, time will be measured in *ticks*. The duration of a tick is defined by the hardware timer on the PS/2; there are 1,193,180 ticks per second.

4.2 Modeling Hardware Interrupts

Each interrupt handler in the application executes in response to a particular hardware interrupt. Furthermore, every application task except the `initiate_send` task executes in response to messages sent by an interrupt handler or another application task that executes, directly or indirectly, in response to a hardware interrupt. Thus, an analysis of the tasks and interrupt handlers in the application must begin with an analysis of the four hardware interrupts listed in Figure 2-1.

The interrupt handler for the first of these interrupts, IRQ0 (*i.e.*, the PS/2 timer), fits directly into the formal model. This interrupt is raised periodically at a rate of 18.2 times per second and is therefore referred to as a *periodic interrupt*. Because it is periodic, successive occurrences of the IRQ0 interrupt are separated by a fixed interval of approximately 55 ms. Thus, the `TIMER` interrupt handler has a well-defined minimum interarrival time. In general, interrupt handlers that execute in response to periodic interrupts can be included in a task system τ as follows: if a handler has an execution cost of e, and the period of the associated periodic interrupt is a, then an interrupt handler (e, a) is included in τ.

The other hardware interrupts, IRQ9, IRQ10, (*i.e.*, the two interrupts from the ActionMedia adapter) and IRQ15 (*i.e.*, the interrupt from the network adapter) do not fit directly into the formal model. One reason is that each of these interrupts can be raised by the hardware in response to an operation initiated by an application task. Consider the IRQ10 interrupt that is raised by the ActionMedia adapter immediately after an `audio_acquire` operation is executed by the `audio` task (see Section 2.5.10). Because this interrupt is generated in response to an operation executed by an application task, it is referred to as a *request-response* interrupt.

The reason that request-response interrupts do not fit directly into the formal model is that the minimum time between successive interrupts can be quite small. Even though the `audio` task is invoked periodically, the time at which the `audio_acquire` operation is executed by the `audio` task depends on where in the interval between its invocation and its deadline the task executes. If two successive invocations of the `audio` task execute, the first completing near its deadline, and the second starting immediately after it is invoked, the length of the minimum interval between the two `audio_acquire` operations (and thus between the two IRQ10 interrupts) can be nearly zero. Figure 4-1 illustrates this. In the figure, the `audio` task is invoked at times s, $s+p$, $s+2p$, etc., and the gray boxes indicate the time at which the invocations execute. The arrows indicate the times at which the IRQ10 interrupts occur.

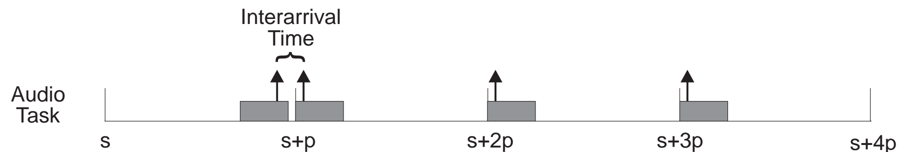

Figure 4-1: Successive Executions of the Audio Task

As a result, the DVI2 interrupt handler cannot be included in the formal model using its true minimum interarrival time. Analysis based on a nearly zero minimum interarrival time would conclude that the processor could spend nearly all of its time executing the interrupt handler. Such analysis would be extremely pessimistic in that, over time, the fraction of processor time spent executing the handler will be equal to the cost of the handler divided by the period of the `audio` task.

However, I can make use of a simple observation in order to incorporate interrupt handlers that execute in response to request-response interrupts into the formal model. Consider four successive invocations of the `audio` task. In the worst case, the end of the first invocation and the start of the third invocation are separated by at least the period of the `audio` task. This is illustrated in Figure 4-2. The DVI2 interrupt handler can be modeled as a pair of interrupt handlers, one representing odd numbered invocations of DVI2, and the other representing even numbered invocations. In general, if we can determine a lower bound on the time between the ith and the $i+n^{th}$ request-response interrupt, then interrupt handlers that execute in response to the interrupt can be included in a task system τ as follows: if a handler has an execution cost of e, and the lower bound on the time between the ith and the $i+n^{th}$ interrupt is c (*i.e.*, $t_{i+n} - t_i \geq c$ for some constant c

where t_i is the time at which the ith interrupt occurs), then n identical interrupt handlers (e,c) are included in τ. A method for determining such bounds is given in the next section.

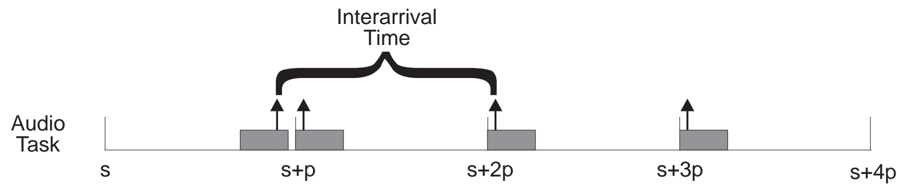

Figure 4-2: Interval Between Odd/Even Pairs of Audio Tasks

Another reason why some hardware interrupts do not fit directly into the formal model is illustrated by the IRQ9 and IRQ15 interrupts. These interrupts are each raised for several reasons. Such interrupts are referred to as *overloaded interrupts*. Consider the IRQ9 interrupt. This interrupt is raised by the ActionMedia hardware for two reasons: to signal the start of a vertical blanking interval (*i.e.*, the VBI logical interrupt) and to signal that a compression operation has completed (*i.e.*, the CC logical interrupt). Because these two events may occur simultaneously, the length of the minimum interval between successive occurrences of the IRQ9 interrupt can be nearly zero. Thus, as with request-response interrupts, an interrupt handler for an overloaded interrupt cannot be included directly in the formal model using its true minimum interarrival time. Rather, an interrupt handler for each logical interrupt can be included in the model. In general, interrupt handlers that execute in response to overloaded interrupts are included in a task system τ as follows: if the handler has an execution cost of e, and the minimum interarrival times of the logical interrupts are $a_1, a_2, ..., a_n$, then n interrupt handlers $(e, a_1), (e, a_2), ..., (e, a_n)$ are included in τ.

4.3 Reasoning about Request-Response Interrupts

In the previous section, I described request-response interrupts and illustrated the difficulty that would arise if such interrupts were included directly in a formal model of an application. I also described a method based on the determination of a lower bound on the time between the occurrences of the ith and the $i+n^{th}$ interrupt that could be used to include such interrupt handlers in a task system. In this section, I develop a technique for determining this lower bound.

I begin with a formal definition of a request-response interrupt. Let T be a task that makes requests to a hardware device to perform an operation. The hardware *responds* to the

request by generating an interrupt I when the requested operation is complete. I is referred to as a request-response interrupt. The task that makes the request is referred to as making a *request for I*. The time required for the hardware device to complete the operation is referred to as the *response time* of I. Let

p be the minimum interarrival time of T.

d be the relative deadline of T.

α be a lower bound on the response time of I.

ω be an upper bound on the response time of I.

r_k be the time at which the kth request for I occurs.

t_k be the time at which the kth instance of I occurs.

Since I is raised in response to each request made by T, the kth occurrence of interrupt I occurs in response to the kth request made by T. If requests are processed sequentially and in order (*i.e.*, servicing of the $k+1^{st}$ request is delayed until the kth request finishes), then the earliest the kth interrupt can occur is α time units after the later of: the time the kth request occurred or the time the k-1st request completes. This property is expressed by equation (4.1).

$$t_k \geq \begin{cases} r_1 + \alpha & \text{if } k = 1 \\ \max\left(r_k + \alpha, t_{k-1} + \alpha\right) & \text{if } k > 1 \end{cases} \quad (4.1)$$

Similarly, the latest the kth interrupt can occur is ω time units after the later of the time the kth request occurred or the time the previous request k-1st completes. This property is expressed by equation (4.2).

$$t_k \leq \begin{cases} r_1 + \omega & \text{if } k = 1 \\ \max\left(r_k + \omega, t_{k-1} + \omega\right) & \text{if } k > 1 \end{cases} \quad (4.2)$$

As shown in the next theorem, equations (4.1) and (4.2) can be used to derive a simple lower bound on the time between occurrences of the ith and the $i+n^{th}$ request-response interrupt.

Theorem 4.1: Let I be a request-response interrupt that occurs in response to requests made by invocations of an application task T. Assume that each invocation of T makes at most one request. Then for all $i > 0$ and for all $n > 0$

$$t_{i+n} - t_i \geq n\alpha \qquad (4.3)$$

Proof: By induction on n.

Basis: Assume that $n = 1$. By equation (4.1)

$$\begin{aligned} t_{i+1} &\geq \max\left(r_{i+1} + \alpha, t_i + \alpha\right) \\ &\geq t_i + \alpha \end{aligned}$$

so $t_{i+1} - t_i \geq \alpha$. Thus, equation (4.3) holds for $n = 1$.

Induction step: Assume equation (4.3) holds for $n = k$. By equation (4.1) and the inductive hypothesis

$$\begin{aligned} t_{i+k+1} - t_i &\geq \max\left(r_{i+k+1} + \alpha, t_{i+k} + \alpha\right) - t_i \\ &\geq t_{i+k} - t_i + \alpha \\ &\geq k\alpha + \alpha \\ &\geq (k+1)\alpha \end{aligned}$$

Thus, equation (4.3) holds for $n = k + 1$. This proves the theorem. □

In practice, the bound given by Theorem 4.1 is not always useful because α is quite small for many request-response interrupts. However, there is a difficulty in obtaining a larger bound. If the maximum response time of I is greater than the maximum rate at which requests are made (*i.e.*, if $\omega > p$), then it is possible for an unbounded number of requests to be queued awaiting service. In this case, the lower bound determined in Theorem 4.1 is a true lower bound. If the lower bound given by Theorem 4.1 is to be improved, one of two constraints must be imposed: either the maximum response time must be less than the period of the task that makes the requests (*i.e.*, $\omega \leq p$), or the number of outstanding requests must be bounded. These will be referred to as *bounded-time interrupts* and *bounded-request interrupts* respectively.

The next theorem improves the bound given in Theorem 4.1 for bounded-time interrupts. Before proving the theorem however, I prove two lemmas. Lemma 4.2 shows that

bounded-time request-response interrupts always occur within $d+\omega$ ticks of the request. Lemma 4.3 uses Lemma 4.2 to derive a lower bound on the interval between request-response interrupts. Finally, Theorem 4.4 combines the bound given in Theorem 4.1 with the bound given in Lemma 4.3.

Lemma 4.2: Let *I* be a bounded-time interrupt that occurs in response to requests made by invocations of an application task *T*. Assume that each invocation of *T* makes at most one request. Let s_i be the time at which the invocation of *T* that makes the ith request for *I* occurs. Then for all $i > 0$

$$t_i \le s_i + d + \omega \tag{4.4}$$

Proof: By induction on *i*.

Basis: Assume $i = 1$. The first request is made at time r_1 by an invocation of *T* that occurred at time s_1. Since the request must occur before the deadline of the task invocation, $r_1 \le s_1 + d$. By equation (4.2), $t_1 \le r_1 + \omega$. Thus, $t_1 \le s_1 + d + \omega$. Thus, equation (4.4) holds for $i = 1$.

Induction step: Assume equation (4.4) holds for $i = k$. The $k+1^{st}$ request for *I* is made by an invocation of *T* that occurred at time s_{k+1} so $r_{k+1} \le s_{k+1} + d$. Because each invocation of *T* makes at most one request, the kth request for *I* must have been made by an invocation of *T* occurring prior to s_{k+1}. Because *T* is sporadic, this invocation must have occurred at or before $s_{k+1} - p$. Thus, $s_k \le s_{k+1} - p$. Because *I* is a bounded-time interrupt, $\omega \le p$. By the inductive hypothesis

$$\begin{aligned} t_k &\le s_k + d + \omega \\ &\le (s_{k+1} - p) + d + \omega \\ &\le s_{k+1} - p + d + p \\ &\le s_{k+1} + d \end{aligned}$$

and thus by equation (4.2),

$$\begin{aligned} t_{k+1} &\le \max(r_{k+1} + \omega, t_k + \omega) \\ &\le \max(s_{k+1} + d + \omega, s_{k+1} + d + \omega) \\ &\le s_{k+1} + d + \omega \end{aligned}$$

Thus, equation (4.4) holds for $i = k+1$. This proves the theorem. □

The next lemma uses Lemma 4.2 to derive a lower bound on the interval between request-response interrupts.

Lemma 4.3: Let I be a bounded-time interrupt that occurs in response to requests made by invocations of an application task T. Assume that each invocation of T makes at most one request. Then for all $i > 0$ and for all $n > 0$

$$t_{i+n} - t_i \geq np - d - (\omega - \alpha)$$

Proof: Assume that the ith request for I is made by an invocation of T that occurs at s_i. By Lemma 4.2, $t_i \leq s_i + d + \delta_{\max}$. Because T is sporadic, for all $n > 0$, $s_{i+n} \geq s_i + np$. Thus, by equation (4.1),

$$\begin{aligned} t_{i+n} &\geq \max(r_{i+n} + \alpha, t_{i+n-1} + \alpha) \\ &\geq r_{i+n} + \alpha \\ &\geq s_{i+n} + \alpha \\ &\geq s_i + np + \alpha \end{aligned}$$

Thus,

$$\begin{aligned} t_{i+n} - t_i &\geq (s_i + np + \alpha) - (s_i + d + \omega) \\ &\geq np - d - (\omega - \alpha) \end{aligned}$$

This proves the lemma. □

Theorem 4.4 combines the bound given in Theorem 4.1 with the bound given in Lemma 4.3.

Theorem 4.4: Let I be a bounded-time interrupt that occurs in response to requests made by invocations of an application task T. Assume that each invocation of T makes at most one request. Then for all $i > 0$ and for all $n > 0$

$$t_{i+n} - t_i \geq \max(np - d - (\omega - \alpha), n\alpha)$$

Proof: The proof is a combination of Theorem 4.1 and Lemma 4.3. By Theorem 4.1,

$$t_{i+n} - t_i \geq n\alpha$$

By Lemma 4.3,

$$t_{i+n} - t_i \geq np - d - (\omega - \alpha)$$

Thus $t_{i+n} - t_i \geq \max(np - d - (\omega - \alpha), n\alpha)$. □

The next theorem improves the bound given in Theorem 4.1 for bounded-request interrupts. Let b be an upper bound on the number of outstanding requests for an interrupt I where at any given time, the number of outstanding requests is defined as the difference between the number of requests that have been made and the number of interrupts that have been generated. If the number of outstanding requests for I is bounded by b, then for all i, $r_{i+b} \geq t_i$.

Before proving the theorem, I prove a lemma. This lemma shows that the interval between bounded-request interrupts is bounded. Theorem 4.6 combines the lower bound given in Theorem 4.1 with the lower bound given in Lemma 4.5.

Lemma 4.5: Let I be a bounded-request interrupt that occurs in response to requests made by invocations of an application task T. Let b be the upper bound on the number of outstanding requests for I. Assume that each invocation of T makes at most one request. Then for all $i > 0$ and for all $m > 0$

$$t_{i+b+m} - t_i \geq mp - d + \alpha$$

Proof: Let $k = i + b + m$ and assume that the kth request for I is made by an invocation of T that occurred at time s_k. Because T is sporadic, for all $m > 0$, $s_{i+b+m} \geq s_{i+b+1} + (m-1)p$. Thus by equation (4.1)

$$\begin{aligned} t_{i+b+m} &\geq \max(r_{i+b+m} + \alpha, t_{i+b+m-1} + \alpha) \\ &\geq r_{i+b+m} + \alpha \\ &\geq s_{i+b+m} + \alpha \\ &\geq s_{i+b+1} + (m-1)p + \alpha \end{aligned} \quad (4.5)$$

Also, because T is sporadic, $s_{i+b} \leq s_{i+b+1} - p$. Since by the definition of a bounded-request interrupt $r_{i+b} \geq t_i$

$$\begin{aligned} t_i &\leq r_{i+b} \\ &\leq s_{i+b} + d \\ &\leq s_{i+b+1} - p + d \end{aligned} \quad (4.6)$$

Combining equations (4.5) and (4.6)

$$\begin{aligned} t_{i+b+m} - t_i &\geq \left(s_{i+b+1} + (m-1)p + \alpha\right) - \left(s_{i+b+1} - p + d\right) \\ &\geq mp - d + \alpha \end{aligned}$$

This proves the lemma. □

Theorem 4.6: Let *I* be a bounded-request interrupt that occurs in response to requests made by invocations of an application task *T*. Let *b* be the upper bound on the number of outstanding requests for *I*. Assume that each invocation of *T* makes at most one request. Then for all $i > 0$ and for all $n > b$

$$t_{i+n} - t_i \geq \max\left((n-b)p - d + \alpha, n\alpha\right)$$

Proof: By Theorem 4.1,

$$t_{i+n} - t_i \geq n\alpha$$

Substituting $n-b$ for *m* in Lemma 4.5,

$$t_{i+n} - t_i \geq (n-b)p - d + \alpha$$

Thus $t_{i+n} - t_i \geq \max\left((n-b)p - d + \alpha, n\alpha\right)$. □

4.4 Determining the Minimum Interarrival Time of Application Tasks

Previously in this chapter, I have discussed the problem of determining the minimum interarrival time of each interrupt handler in the application. In this section, I discuss the problem of determining the minimum interarrival time of each of the application tasks. There are three classes of application tasks: those invoked by interrupt handlers, those invoked by other application tasks, and those invoked by the YARTOS periodic invocation mechanism. The problem of determining minimum interarrival times for each of these classes is discussed below.

4.4.1 Application Tasks Invoked by Interrupt Handlers

Consider an application task that is invoked by a message sent from a logical interrupt handler. As described in Chapter 2, the first activity performed by an interrupt handler is

to determine the current time; this time defines the logical time at which messages are sent from that interrupt handler. Thus, the minimum interarrival time of the application task is determined by the minimum interval between the time measurements by successive executions of the logical interrupt handler. Figure 4-3 illustrates the situation that defines the minimum interarrival time of the application task. In this figure, the dark gray boxes represent the execution of the interrupt handler, the light gray box represents an interrupt handler that is delayed (*e.g.*, by executions of higher priority interrupts), and the arrows indicate the logical arrival time of a message sent to the task. Thus, the two logical interrupts are separated by the minimum interarrival time of the interrupt, execution of the first interrupt handler is delayed (*e.g.*, by executions of higher priority interrupts), and execution of the second interrupt handler occurs immediately.

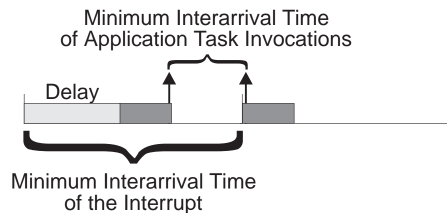

Figure 4-3: Minimum Interarrival Time of Application Task Invocations

Thus, to determine the minimum interarrival time of an application task that is invoked by a message from a logical interrupt handler, it is necessary to know the maximum time for which an interrupt handler can be delayed before it completes execution. For an interrupt handler I, this time will be denoted E_I. Thus, if I is an interrupt handler with a minimum interarrival time of a that sends a message to an application task, that task will have a minimum interarrival time of $a - E_I$.

The problem of determining the maximum time for which an interrupt handler can be delayed before it completes execution is similar to that of determining an upper bound on the completion time of task invocations in a set of sporadic tasks that execute with arbitrary fixed priorities. In [12], Harbour, Klein, and Lehoczky discuss the problem of determining this bound for sets of periodic tasks with arbitrary fixed priorities[6].

---

[6]The authors actually discuss a more general problem, in which a task consists of phases, each of which can have a different fixed priority and an individual deadline. For my purposes here, I only require tasks with a single phase. Thus, I am simplifying the results presented by the authors by considering only this special case.

Fortunately, their analysis of the completion time of task invocations does not depend on the periodicity of tasks. Thus, we can apply their analysis framework in our problem.

To use this framework, we must determine several values for each interrupt handler. For an interrupt handler *I*, let

MP_I be the set of interrupt handlers with priority greater than that of *I*.

B_I be the "blocking term"

The *MP* term is used to determine the amount of time the execution of an interrupt handler can be delayed by the execution of higher-priority interrupt handlers. These delays include both the delay added by the interrupt controller that delays delivery of an interrupt until all higher-priority interrupts have been serviced, and delays added when execution of the interrupt handler is preempted by a higher-priority interrupt handler.

The blocking term is a value that captures the maximum time that the start of an interrupt handler can be delayed for reasons other than the execution of higher priority interrupt handlers. As described in Chapter 2, several things other than higher-priority handlers can delay the start of an interrupt handler. First, the YARTOS kernel uses a CPU flag to disable all interrupts. A logical interrupt handler can be delayed for another reason: it cannot begin executing until any outstanding logical interrupts overloaded on the same interrupt are serviced[7].

The maximum delay incurred by an interrupt handler due to disabled interrupts occurs when interrupts are disabled one time unit before the interrupt occurs and remain disabled for the maximum possible time. The maximum delay incurred due to the execution of an overloaded interrupt handler occurs when the overloaded handler begins execution one time unit before the interrupt and executes for a time equal to its maximum execution cost. However, an interrupt handler can be delayed either because the interrupt occurs while interrupts are disabled, or because the interrupt occurs while another overloaded interrupt is being serviced. It cannot be delayed by both. Thus, if the following terms represent these values:

---

[7]There is an additional reason an interrupt handler may be delayed, namely that an application can use the interrupt controller to mask individual interrupts. However, this mechanism is not used on the acquisition-side, so I do not consider it here.

kernel the maximum time interrupts are continuously disabled

$overload_I$ the maximum cost of interrupt handlers overloaded on the same interrupt

then the blocking term is given by

$$B_I = \max\left(kernel - 1, overload_I - 1, 0\right)$$

With the blocking term defined, I can determine the upper bound on the time required to complete execution of the interrupt handler using the Harbour, et al., results. For an interrupt handler *I* with execution cost *e*, this is given by

$$E_I = \min\left(t > 0 \middle| \left(B_I + \sum_{j \in MP_I} \lceil t/a_j \rceil e_j + e \right) = t \right) \tag{4.7}$$

The upper bound on delay given by equation (4.7) can now be used to determine the minimum interarrival time of an application task that is invoked by a message from a logical interrupt handler. If *I* is an interrupt handler with a minimum interarrival time of *a* that sends a message to an application task, then that task can be included in a task system τ with a minimum interarrival time of $a - E_I$.

4.4.2 Application Tasks Invoked by Other Tasks

Next, I address the problem of determining the minimum interarrival time of a task that is invoked by a message sent from another application task. An observation made by Jeffay in [23] is helpful in simplifying this problem. Consider a task invocation, called the *receiver*, that is invoked by a message sent to it from another task invocation, called the *sender*. If it is not possible for the receiver to preempt the sender, then the logical arrival time of the receiver can be set to the logical arrival time of the sender. This is the rule used to determine logical arrival times (and thus deadlines) in YARTOS.

Thus, in the analysis performed in this chapter, the minimum interarrival time of a receiving task (*i.e.*, a lower bound on the length of the interval between successive logical arrival times of the task) is defined by the minimum interarrival time of the sending task. Assume the sending task has a minimum interarrival time of *p*. If the sending task sends a message to the receiving task each time it executes, then the receiving task can be included in a task system τ with a minimum interarrival time of *p*. More generally, if the sending

task sends a message every n^{th} time it executes, then the receiving task can be included in a task system τ with a minimum interarrival time of np.

4.4.3 YARTOS Periodic Invocations

Next, I address the problem of determining the minimum interarrival time of a task that is invoked by messages sent directly from YARTOS. As described in Chapter 2, an application task declaration can specify that YARTOS should periodically send messages directly to the task (*e.g.*, the `initiate_send` task described in Section 2.5.8). Abstractly, these messages are generated periodically. In practice however, the messages are only approximately periodic.

Consider an application task T that is to be periodically invoked by YARTOS with period p. Assume that the first message generated by YARTOS is sent to T at time t. In this case, the YARTOS will attempt to generate a message to T as soon as possible after times $t + kp$ for $k \geq 1$. If the processor is idle (*i.e.*, there are no tasks or interrupt handlers ready for execution), the YARTOS kernel continuously checks to see if a new message should be generated; if so a message is sent to T. Otherwise, each time an interrupt handler or application task completes execution, the YARTOS kernel checks to see if a new message should be generated.

Thus, the minimum interval between successive invocations of T will be somewhat smaller than the period p, depending on the maximum time that can elapse between the time YARTOS should have generated a message and the time it actually does generate a message. However, if each invocation of T is assigned a logical arrival time equal to the time at which it should have received the message, then the interval between the logical arrival times of successive invocations of T will be exactly equal to p.

Assume that YARTOS should have sent a message to a task T at time t, but is unable to send the message until time t' due to the fact that processor was not idle and no application task or interrupt handler completed execution in the interval $[t, t']$. If it is assumed that the scheduling discipline would not have chosen to execute the task invocation anywhere in the interval $[t, t']$, then the delay in actually sending the message will have no effect on the execution of the system. Thus, if this assumption can be enforced, then the task invocation can be assigned a logical arrival time of t without affecting the feasibility of the task system. One method for enforcing this assumption is to ensure that the application task shares an individual resource with each task in the task

system. As a result, under the YARTOS scheduling discipline, it can never preempt another application task, thus ensuring that it would not have executed in $[t, t']$.

Thus, if an application task that receives messages periodically from YARTOS at a period p shares a resource with each task in a task system τ , then the application task can be included in τ with a minimum interarrival time of p. This is the approach that will be used to include the `initiate_send` task in the analysis performed in the remainder of the chapter.

4.5 Feasibility of the Application

In Chapter 2, Figure 2-20 illustrated the software architecture of the video conferencing application, showing the various hardware interrupts, interrupt handlers, application tasks, and resources as well as the resource usage of each task and the message passing channels between interrupt handlers and tasks (or between tasks and tasks). Figure 4-4 illustrates an alternative view of the architecture of the application. It shows logical interrupts and message channels, as well as *device requests* that indicate that a task or handler executes an operation that leads to a request-response interrupt. Logical interrupts are labeled by either a period, indicating that the logical interrupt is generated periodically, or by a response time, indicating that the logical interrupt is a request-response interrupt with the indicated constraints on its response time (*e.g.*, the `CC` logical interrupt is a request-response interrupt with a lower bound on response time of 22 ms., and an upper bound on response time of 28 ms.). Some message passing channels are labeled with a rate, indicating that messages are sent along that channel 1 out of every N times the sender is executed; a channel without an indicated rate is assumed to have a rate of 1/1.

In the remainder of this section, I use the principles laid out earlier in the chapter to determine execution costs and minimum interarrival times for each of the interrupt handlers and application tasks in the system. From these values, I construct a task system τ as defined in Section 3.2. I then illustrate the results of applying the feasibility test given by Theorem 3.5 to the resulting task system.

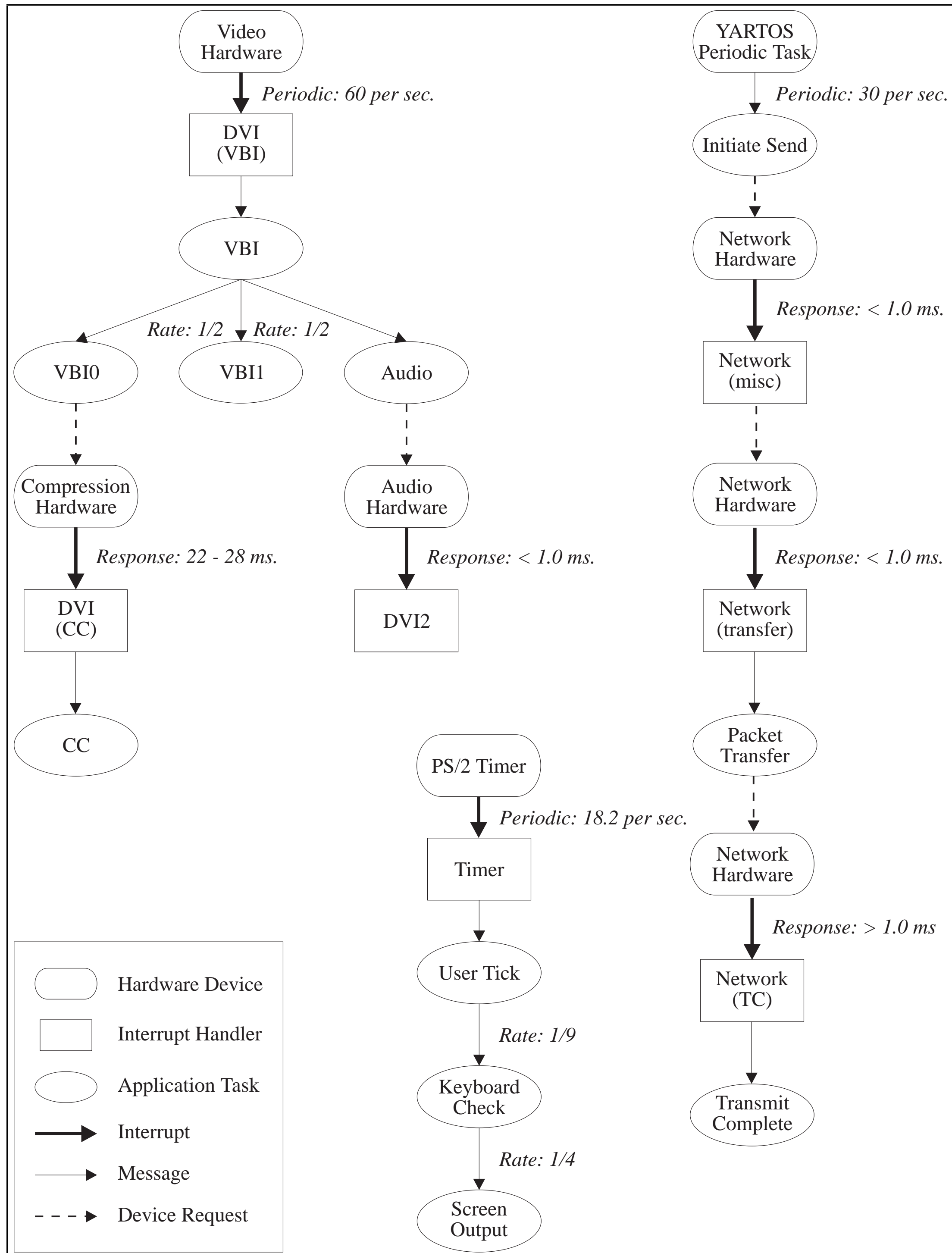

Figure 4-4: An Alternative View of the Acquisition-Side Architecture

4.5.1 Estimating Execution Costs

I begin by estimating an upper bound on execution time for each interrupt handler and application task in the system. In general, obtaining a reasonable upper bound on these costs is a very difficult problem. One approach used by Park and Shaw [39] used a modified compiler to instrument tasks in order to provide data for an analytic approach defined by Shaw in [48]. Such efforts are beyond the scope of the dissertation. However, the set of tasks and interrupt handlers in the video conferencing application are quite simple. When the application is executing normally (*i.e.*, successfully acquiring and transmitting every frame), most tasks execute a fixed sequence of instructions (*i.e.*, to move buffers between shared queues and to control hardware operations). Furthermore, this sequence of instructions is the longest path through the task.

Task or Handler	Type	Execution Cost (ticks)	Average Cost (ticks)
DVI	Handler	398	278
DVI2	Handler	218	177
TIMER	Handler	303	235
NETWORK	Handler	464	296
VBI	Task	472	364
VBI0	Task	1213	720
VBI1	Task	904	820
Audio	Task	1102	1010
CC	Task	603	464
Transmit Complete	Task	279	210
Packet Transfer	Task	10262	9614
Initiate Send	Task	1004	918
User Tick	Task	212	126
Keyboard Check	Task	800	634
Screen Output	Task	206	189

Figure 4-5: Execution Costs (1193 ticks per ms.)

Thus, for each of the application tasks and interrupt handlers in the application, it is reasonable to use empirical measurements of the performance of the application under normal conditions to obtain an estimate of an upper bound on execution time. To support measurement of execution costs, the YARTOS kernel has been instrumented to record

execution time statistics. For each task and interrupt handler, YARTOS records a histogram of execution times along with the minimum, maximum, and average execution times. To obtain the upper bounds on execution cost used here, I ran the conferencing application for 10 minutes (*i.e.*, processing 36000 audio frames and 18000 video frames) and chose the maximum recorded execution time for each task and interrupt handler. The resulting estimates of execution cost are presented in Figure 4-5. For comparison, I also provide the average execution cost for each task and interrupt handler; in most cases the average and the maximum cost are quite similar.

In the execution of the conferencing application, I also determined two other values. First, I determined an upper bound on the length of an interval in which the processor can execute continuously with interrupts disabled; during the run, the longest interval during which the processor executed continuously with interrupts disabled was 160 ticks (*i.e.*, the value of *kernel* used in calculating block terms will be 160). I also determined the actual processor utilization of the system, defined as the fraction of the time during which interrupt handlers and application tasks were executed; the resulting utilization measurement was approximately 47%.

4.5.2 Estimating Minimum Interarrival Times

The next step in the development of the formal model is to use the principles laid out earlier in the chapter to determine minimum interarrival times for each of the logical interrupt handlers and application tasks in the system. Because the minimum interarrival times of tasks and handlers often depend on those assigned to other tasks or handlers, I proceed by traversing the graph given in Figure 4-4 in topological order beginning with the high-priority periodic interrupts (see Section 2.3). For each hardware interrupt, I determine the maximum time required to complete execution of the interrupt handler. For each task and logical interrupt handler in the graph, I determine its minimum interarrival time. For each request-response interrupt in the graph, I determine the number of copies of the interrupt handler that must be included in the formal model. Note that throughout this discussion, time will be measured in ticks (recall that there are 1,193,180 ticks per second).

- **TIMER**. IRQ0 is a periodic interrupt raised every 65,536 ticks. Thus, the minimum interarrival time of the `TIMER` interrupt handler is

$$a_{\text{TIMER}} = 65,536$$

IRQ0 is the highest priority interrupt and is not overloaded. Thus

$$B_{\text{TIMER}} = kernel - 1 = 159$$
$$MP_{\text{TIMER}} = \varnothing$$

Therefore, the maximum time required to complete execution of the interrupt hander is given by equation (4.7)

$$\begin{aligned} E_{\text{TIMER}} &= \min\left(t > 0 \middle| B_{\text{TIMER}} + e_{\text{TIMER}} = t\right) \\ &= \min\left(t > 0 \middle| 159 + 303 = t\right) \\ &= 462 \end{aligned}$$

- **user_tick**. This task is invoked by messages from the `TIMER` interrupt handler. Thus, its minimum interarrival time is

$$p_{\text{user_tick}} = a_{\text{TIMER}} - E_{\text{TIMER}} = 65{,}074$$

- **keyboard_check**. This task is invoked every 9th execution of the `user_tick` task. Thus, its minimum interarrival time is

$$p_{\text{keyboard_check}} = 9 p_{\text{user_tick}} = 585{,}666$$

- **screen_output**. This task is invoked every 4th execution of the `keyboard_check` task. Thus, its minimum interarrival time is

$$p_{\text{screen_output}} = 4 p_{\text{keyboard_check}} = 2{,}342{,}664$$

- **DVI/VBI**. The VBI logical interrupt is a periodic interrupt raised by the ActionMedia hardware every 19,886 time units (60 times per sec.). Thus, the minimum interarrival time of the logical interrupt handler is

$$a_{\text{DVI/VBI}} = 19{,}886$$

The VBI logical interrupt corresponds to the IRQ9 hardware interrupt, which is an overloaded hardware interrupt raised in response to both VBI logical interrupts and CC logical interrupts. IRQ9 is lower priority than IRQ0. Thus

$$\begin{aligned} overload_{\text{DVI/VBI}} &= e_{\text{DVI}} = 398 \\ B_{\text{DVI/VBI}} &= \max\left(kernel - 1, overload_{\text{DVI/VBI}} - 1\right) = 397 \\ MP_{\text{DVI/VBI}} &= \{\text{TIMER}\} \end{aligned}$$

Therefore, the maximum time required to complete execution of the interrupt hander is given by equation (4.7)

$$
\begin{aligned}
E_{\text{DVI/VBI}} &= \min\left(t > 0 \,\middle|\, B_{\text{DVI/VBI}} + \left\lceil \frac{t}{a_{\text{TIMER}}} \right\rceil e_{\text{TIMER}} + e_{\text{DVI}} = t \right) \\
&= \min\left(t > 0 \,\middle|\, 397 + \left\lceil \frac{t}{65,536} \right\rceil 303 + 398 = t \right) \\
&= 1,098
\end{aligned}
$$

- **VBI**. This task is invoked by messages from the DVI/VBI logical interrupt handler. Thus, its minimum interarrival time is

$$p_{\text{VBI}} = a_{\text{DVI/VBI}} - E_{\text{DVI/VBI}} = 18,788$$

- **VBI0**. This task is invoked every 2nd execution of the `VBI` task. Thus, its minimum interarrival time is

$$p_{\text{vbi0}} = 2p_{\text{vbi}} = 37,576$$

- **DVI/CC**. The CC logical interrupt is a bounded-time request-response interrupt requested by the VBI0 task. Lower and upper bounds on the response time of requests for CC interrupts were given in Figure 4-4. Converted to ticks, these bounds are:

$$
\begin{aligned}
\alpha_{\text{CC}} &= 26,250 \\
\omega_{\text{CC}} &= 33,408
\end{aligned}
$$

Theorem 4.4 can now be used to determine the minimum interarrival time. For all $n \geq 1$, this theorem defines a lower bound on the length of the interval between the ith and $i+n^{\text{th}}$ occurrences of the interrupt. Choosing n to be the smallest possible value for which the resulting minimum interarrival time is reasonable, in this case $n = 1$, the minimum interarrival time of the interrupt handler is

$$a_{\text{DVI/CC}} = p_{\text{VBI0}} - d_{\text{VBI0}} - (\omega_{\text{CC}} - \alpha_{\text{CC}}) = 12,520$$

By definition, each overloaded interrupt handler has the same completion time, so

$$E_{\text{DVI/CC}} = E_{\text{DVI/VBI}} = 1,098$$

- **CC**. This task is invoked by messages from the DVI/CC interrupt handler. Thus, its minimum interarrival time is

$$p_{\text{CC}} = a_{\text{DVI/CC}} - E_{\text{DVI/CC}} = 11,422$$

- **VBI1**. This task is invoked every 2nd execution of the `VBI` task. Thus, its minimum interarrival time is

$$p_{\text{VBI1}} = 2p_{\text{VBI}} = 37,576$$

- **audio**. This task is invoked by the `VBI` task every time it executes. Thus, its minimum interarrival time is given by

$$p_{\text{audio}} = p_{\text{VBI}} = 18,788$$

- **DVI2**. The DVI2 interrupt is a bounded-time request-response interrupt requested by the `audio` task. Lower and upper bounds on the response time of requests for DVI2 interrupts were given in Figure 4-4:

$$\alpha_{\text{DVI2}} = 0$$
$$\omega_{\text{DVI2}} = 1,193$$

Again, choosing the n in Theorem 4.4 to be the smallest possible value for which the resulting minimum interarrival time is reasonable, in this case $n = 2$, the minimum interarrival time of the interrupt handler is

$$a_{\text{DVI2}} = 2p_{\text{audio}} - d_{\text{audio}} - (\omega_{\text{DVI2}} - \alpha_{\text{DVI2}}) = 18,485$$

Because this minimum interarrival time is based on the choice of $n = 2$, two copies of the DVI2 interrupt handler are included in the formal model. As a result, DVI2 is an overloaded interrupt (*i.e.*, each of the two copies is overloaded with the other). Because this interrupt handler executes in response to the IRQ10 hardware interrupt, it has lower priority than TIMER, DVI/VBI and DVI/CC. Thus

$$overload_{\text{DVI2}} = e_{\text{DVI2}} = 218$$
$$B_{\text{DVI2}} = \max(kernel - 1, overload_{\text{DVI2}} - 1) = 217$$
$$MP_{\text{DVI2}} = \{\text{Timer, DVI/VBI, DVI/CC}\}$$

Therefore, the completion time is given by

$$E_{\text{DVI2}} = \min\left(t > 0 \middle| B_{\text{DVI2}} + \left\lceil \frac{t}{a_{\text{TIMER}}} \right\rceil e_{\text{TIMER}} + \left\lceil \frac{t}{a_{\text{DVI/VBI}}} \right\rceil e_{\text{DVI}} + \left\lceil \frac{t}{a_{\text{DVI/CC}}} \right\rceil e_{\text{DVI}} + e_{\text{DVI2}} = t \right)$$
$$= \min\left(t > 0 \middle| 217 + \left\lceil \frac{t}{65{,}536} \right\rceil 303 + \left\lceil \frac{t}{19{,}886} \right\rceil 398 + \left\lceil \frac{t}{12{,}520} \right\rceil 398 + 218 = t \right)$$
$$= 1{,}534$$

- **initiate_send**. This task is invoked periodically by YARTOS with a period of 39,773 ticks. This task shares a resource with each application task, so its minimum interarrival time is simply

$$p_{\text{initiate_send}} = 39{,}773$$

- **NETWORK/MISC**. This logical interrupt is a bounded-time request-response interrupt requested by the `initiate_send` task. Lower and upper bounds on the response time of requests for this interrupt were given in Figure 4-4:

$$\alpha_{\text{MISC}} = 0$$
$$\omega_{\text{MISC}} = 1{,}193$$

Again, choosing the n in Theorem 4.4 to be the smallest possible value for which the resulting minimum interarrival time is reasonable, in this case $n = 2$, the minimum interarrival time of the interrupt handler is

$$a_{\text{NETWORK/MISC}} = 2 p_{\text{initiate_send}} - d_{\text{initiate_send}} - (\omega_{\text{MISC}} - \alpha_{\text{MISC}}) = 54{,}489$$

Because this minimum interarrival time is based on the choice of $n = 2$, two copies of the NETWORK/MISC interrupt handler are included in the formal model. As a result, NETWORK/MISC is an overloaded interrupt. Because this interrupt handler executes in response to the IRQ15 hardware interrupt, it has lower priority than TIMER, DVI/VBI, DVI/CC, and DVI2. Thus

$$overload_{\text{NETWORK/MISC}} = e_{\text{NETWORK}} = 464$$
$$B_{\text{NETWORK/MISC}} = \max(kernel - 1, overload_{\text{NETWORK/MISC}} - 1) = 463$$
$$MP_{\text{NETWORK/MISC}} = \{\text{Timer, DVI/VBI, DVI/CC, DVI2}\}$$

Therefore, the completion time is given by

$$E_{\text{NETWORK/MISC}} = \min\left(t > 0 \,\middle|\, \begin{array}{l} B_{\text{NETWORK/MISC}} + \left\lceil \dfrac{t}{a_{\text{TIMER}}} \right\rceil e_{\text{TIMER}} + \left\lceil \dfrac{t}{a_{\text{DVI/VBI}}} \right\rceil e_{\text{DVI}} + \\ \left\lceil \dfrac{t}{a_{\text{DVI/CC}}} \right\rceil e_{\text{DVI}} + 2\left\lceil \dfrac{t}{a_{\text{DVI2}}} \right\rceil e_{\text{DVI2}} + e_{\text{NETWORK}} = t \end{array} \right)$$

$$= \min\left(t > 0 \,\middle|\, \begin{array}{l} 463 + \left\lceil \dfrac{t}{65,536} \right\rceil 303 + \left\lceil \dfrac{t}{19,886} \right\rceil 398 + \\ \left\lceil \dfrac{t}{12,520} \right\rceil 398 + 2\left\lceil \dfrac{t}{18,485} \right\rceil 218 + 464 = t \end{array} \right)$$

$$= 2,462$$

- **NETWORK/XFER**. This logical interrupt is a bounded-time request-response interrupt requested by the NETWORK/MISC interrupt handler. I have not yet addressed the question of determining the minimum interarrival time of such an interrupt. However, because an upper bound on the completion time of the NETWORK/MISC handler is known, the minimum interarrival time of the NETWORK/XFER interrupt handler can be determined using a technique similar to that used to determine the minimum interarrival time of a task invoked by an interrupt handler in Section 4.4.1.

  The minimum interval between successive NETWORK/XFER interrupts occurs when two NETWORK/MISC interrupts are separated by their minimum interarrival time, the first makes the request immediately before it completes execution, and the second makes the request immediately. Lower and upper bounds on the response time of requests for the NETWORK/XFER interrupt were given in Figure 4-4:

$$\alpha_{\text{XFER}} = 0$$
$$\omega_{\text{XFER}} = 1,193$$

  Again, choosing the n in Theorem 4.4 to be the smallest possible value for which the resulting minimum interarrival time is reasonable, in this case $n = 1$, the minimum interarrival time of the interrupt handler is

$$a_{\text{NETWORK/XFER}} = a_{\text{NETWORK/MISC}} - E_{\text{NETWORK/MISC}} - (\omega_{\text{XFER}} - \alpha_{\text{XFER}}) = 50,834$$

  Also, since two copies of the NETWORK/MISC interrupt handler are included in the formal model, two copies of the NETWORK/XFER interrupt handler must also be included in the formal model.

  By definition, each overloaded interrupt handler has the same completion time, so

$$E_{\text{NETWORK/XFER}} = E_{\text{NETWORK/MISC}} = 2,462$$

- **packet_transfer**. This task is invoked by messages from the NETWORK/XFER interrupt handler. Thus, its minimum interarrival time is

  $$p_{\text{packet_transfer}} = a_{\text{NETWORK/XFER}} - E_{\text{NETWORK/XFER}} = 48,372$$

  Also, since two copies of the NETWORK/XFER interrupt handler are included in the formal model, two copies of the `packet_transfer` task must also be included in the formal model.

- **NETWORK/TC**. This logical interrupt is a bounded-request request-response interrupt that is indirectly requested by the `initiate_send` task. Because the `initiate_send` does not initiate a new network transmission until the previous transmission completes, the task enforces a bound of one outstanding transmit request. A lower bound on the response time of requests for this interrupt was given in Figure 4-4. Thus,

  $$b = 1$$
  $$\alpha_{\text{TC}} = 1,193$$

  Because this is a bounded-request interrupt, Theorem 4.6 can now be used to determine the minimum interarrival time. Choosing the n in the theorem to be the smallest possible value for which the resulting minimum interarrival time is reasonable, in this case $n = 3$, the minimum interarrival time of the interrupt handler is

  $$a_{\text{NETWORK/TC}} = (3 - b)p_{\text{initiate_send}} - d_{\text{initiate_send}} + \alpha_{\text{TC}} = 56,875$$

  Because this minimum interarrival time is based on the choice of $n = 3$, three copies of the NETWORK/TC interrupt handler are included in the formal model.

  By definition, each overloaded interrupt handler has the same completion time, so

  $$E_{\text{NETWORK/TC}} = E_{\text{NETWORK/MISC}} = 2,462$$

- **transmit_complete**. This task is invoked by messages from the NETWORK/TC interrupt handler. Thus, its minimum interarrival time is

  $$p_{\text{transmit_complete}} = a_{\text{NETWORK/TC}} - E_{\text{NETWORK/TC}} = 54,413$$

Also, since three copies of the NETWORK/TC interrupt handler are included in the formal model, three copies of the `transmit_complete` task must also be included in the formal model.

4.5.3 Using the Feasibility Test

Figure 4-6 summarizes the costs and minimum interarrival times for each logical interrupt handler in the system, as well as the number of copies that should be included in the model, and the upper bound on completion time determined above.

Name	Cost	Interarrival Time	Copies	Completion Time
TIMER	303	65,536	1	462
DVI/VBI	398	19,886	1	1,098
DVI/CC	398	12,520	1	1,098
DVI2	218	18,485	2	1,534
NETWORK/MISC	464	54,489	2	2,462
NETWORK/XFER	464	50,834	2	2,462
NETWORK/TC	464	56,875	3	2,462

Figure 4-6: Summary of Interrupt Handlers (time in ticks)

Figure 4-7 summarizes the costs and minimum interarrival times for each application task in the system, as well as the number of copies that should be included in the model, the relative deadline of the task, and the minimum deadline among all the tasks that share a resource with the task (labeled "minimum deadline").

Name	Cost	Interarrival Time	Relative Deadline	Minimum Deadline	Copies
user tick	212	65,074	39,773	23,864	1
keyboard	800	585,666	39,773	23,864	1
screen output	206	2,342,664	39,773	23,864	1
VBI	472	18,788	17,898	17,898	1
VBI0	1,213	37,576	17,898	9,545	1
CC	603	11,422	9,545	9,545	1
VBI1	904	37,576	17,898	17,898	1
audio	1,102	18,788	17,898	17,898	1
initiate send	1,004	39,773	23,864	9,545	1
packet transfer	10,262	48,372	39,773	23,864	2
TC	279	54,413	39,773	9,545	3

Figure 4-7: Summary of Application Tasks (time in ticks)

The information given in Figures 4-6 and 4-7 can now be used to define the formal model of the application. Let τ be a task system with 12 interrupt handlers $\{I_1,\ldots,I_{12}\}$, 14 application tasks $\{T_1,\ldots,T_{14}\}$, and 21 resources $\{R_1,\ldots,R_{21}\}$. Figures 4-8, 4-9, and 4-10 define each of the interrupt handlers, application tasks, and resources.

Name	e	a	Notes
I_1	303	65,536	TIMER
I_2	398	19,886	DVI/VBI
I_3	398	12,520	DVI/CC
I_4	218	18,485	DVI2 (2 copies)
I_5	218	18,485	
I_6	464	54,489	NETWORK/MISC (2 copies)
I_7	464	54,489	
I_8	464	50,834	NETWORK/XFER (2 copies)
I_9	464	50,834	
I_{10}	464	56,875	NETWORK/TC (3 copies)
I_{11}	464	56,875	
I_{12}	464	56,875	

Figure 4-8: Formal Definitions of the Interrupt Handlers

Name	c	U	d	p	Notes
T_1	212	$\{R_1\}$	39,773	65,074	user tick
T_2	800	$\{R_2\}$	39,773	585,666	keyboard check
T_3	206	$\{R_3\}$	39,773	2,342,664	screen output
T_4	472	$\{R_4\}$	17,898	18,788	VBI
T_5	1,213	$\{R_5,R_{15},R_{16},R_{17},R_{18}\}$	17,898	37,576	VBI0
T_6	603	$\{R_6,R_{17},R_{18},R_{21},\}$	9,545	11,422	CC
T_7	904	$\{R_7,R_{15},R_{16}\}$	17,898	37,576	VBI1
T_8	1,102	$\{R_8,R_{19},R_{20}\}$	17,898	18,778	audio
T_9	1,004	$\{R_1,R_2,R_3,R_4,$ $R_5,R_6,R_7,R_8,$ $R_9,R_{10},R_{11},R_{12},$ $R_{13},R_{14},R_{20},R_{21}\}$	23,864	39,773	initiate send
T_{10}	10,262	$\{R_9\}$	39,773	48,372	packet transfer (2 copies)
T_{11}	10,262	$\{R_{10}\}$	39,773	48,372	
T_{12}	279	$\{R_{11},R_{14},R_{17},R_{19}\}$	39,773	54,413	TC (3 copies)
T_{13}	279	$\{R_{11},R_{14},R_{17},R_{19}\}$	39,773	54,413	
T_{14}	279	$\{R_{11},R_{14},R_{17},R_{19}\}$	39,773	54,413	

Figure 4-9: Formal Definitions of the Application Tasks

Name	Notes
R_1 - R_{13}	Implicit resources for `initiate_send`
R_{14}	`transmit_queue`
R_{15}	`next_digitize` queue
R_{16}	`compress_source` queue
R_{17}	pool of free compress buffers
R_{18}	`compress_sink` queue
R_{19}	pool of free audio buffers
R_{20}	`transmit_audio` queue
R_{21}	`transmit_video` queue

Figure 4-10: Formal Definitions of the Resources

Recall Theorem 3.5. The achievable processor utilization is defined as

$$\Psi_\tau = \sum_{i=1}^{n} \frac{c_i}{p_i} + \sum_{i=1}^{m} \frac{e_i}{a_i}$$

Let

$$B_\tau = \frac{\sum_{i=1}^{m} e_i + \sum_{i=1}^{n} c_i}{1 - \Psi_\tau}$$

and let $P = \{kp_i + d_i \mid kp_i + d_i \le B_\tau \wedge k \ge 0 \wedge 1 \le i \le n\} \cup \{0\}$. Then, if $\Psi_\tau < 1$, τ will be feasible if the following two conditions hold.

1) $\forall L, L \in P$

$$L - f(L) \ge \sum_{i=1}^{n} \delta_i(L) \cdot c_i$$

2) $\forall i, 1 \le i \le n, \forall L, D_i < L < d_i$

$$L - f(L) \ge c_i + \sum_{j=1}^{n} \delta_j(L-1) \cdot c_j$$

For this task system, the achievable processor utilization is:

$$\Psi_\tau = 0.8023$$

and the upper bound for which Condition 1 of the feasibility test must be checked is

$$B_\tau = 165{,}213$$

The result of checking the first condition is shown in Figure 4-11. This graphs the function

$$C_1(L) = L - f(L) - \sum_{i=1}^{n} \delta_i(L) \cdot c_i$$

for all $0 \le L \le B_\tau$. It is the case that Condition 1 holds only if $C_1(L)$ is at least 0 throughout this interval (although it is sufficient to test the condition only at multiples of the minimum interarrival times of tasks).

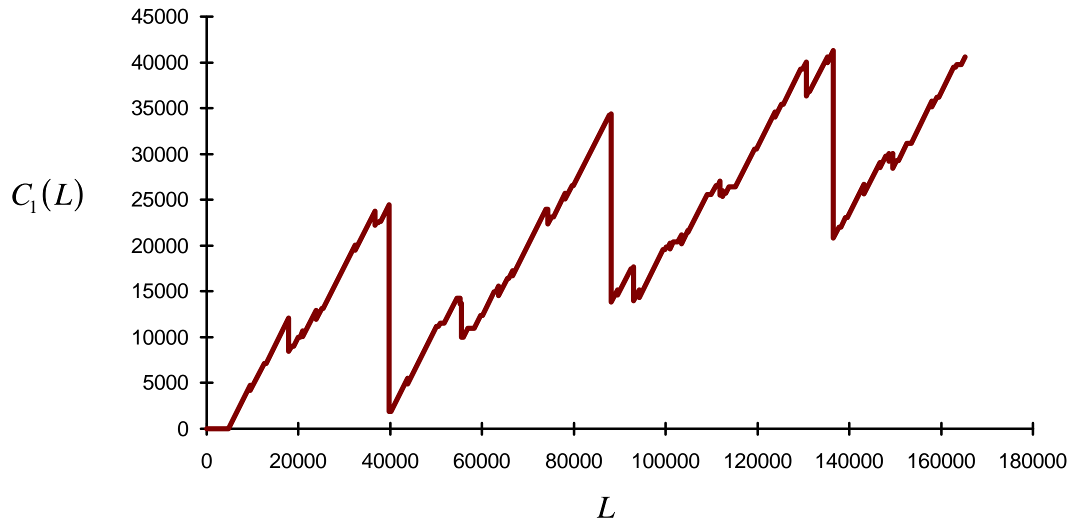

Figure 4-11: Graph of Condition 1 ($0 \le L \le 165,213$)

Condition 2 must be tested for each application task in the system. However, for a number of tasks, the relative deadline of the task is equal to the minimum deadline of the set of tasks it shares resources with (*i.e.*, $d_i = D_i$). As a result, for these tasks, the range of L in Condition 2 is void, and thus the condition holds trivially. This is the case for the `VBI`, `CC`, `VBI1`, and `audio` tasks. For the remaining tasks, Figures 4-12 through 4-18 graph the function

$$C_2(i,L) = L - f(L) - c_i - \sum_{j=1}^{n} \delta_j(L-1) \cdot c_j$$

for all $D_i < L < d_i$. It is the case that Condition 2 holds only if $C_2(i,L)$ is always greater than or equal to 0 in this interval.

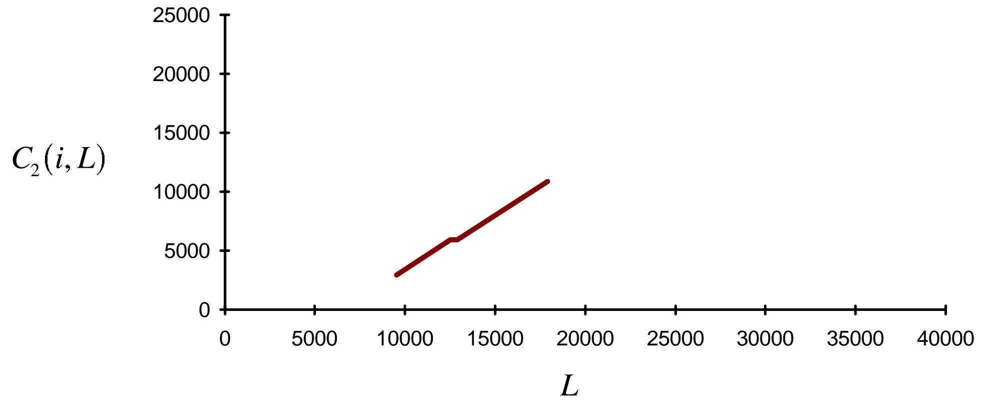

Figure 4-12: Graph of Condition 2 for Vbi0 Task ($9,545 < L < 17,898$)

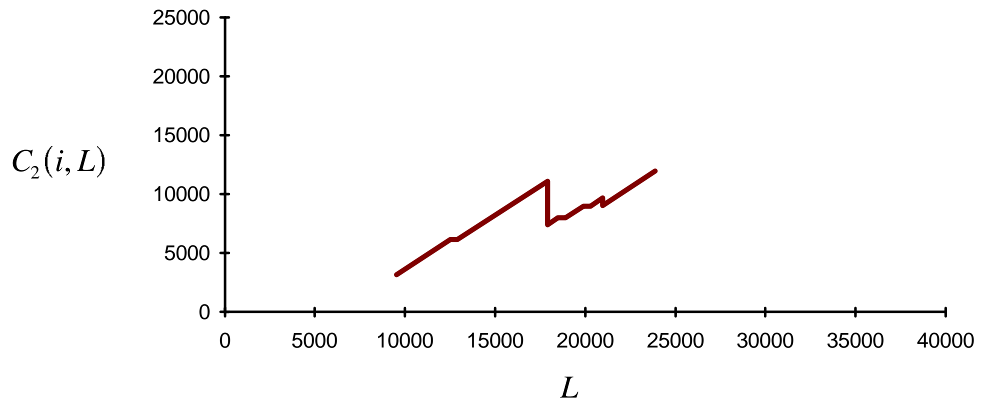

Figure 4-13: Graph of Condition 2 for Initiate Send Task ($9,545 < L < 23,864$)

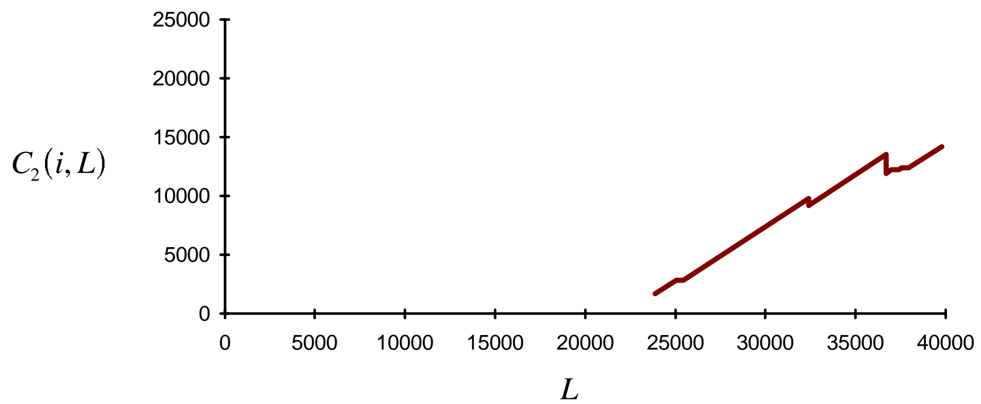

Figure 4-14: Graph of Condition 2 for Packet Transfer Task ($23,864 < L < 39,773$)

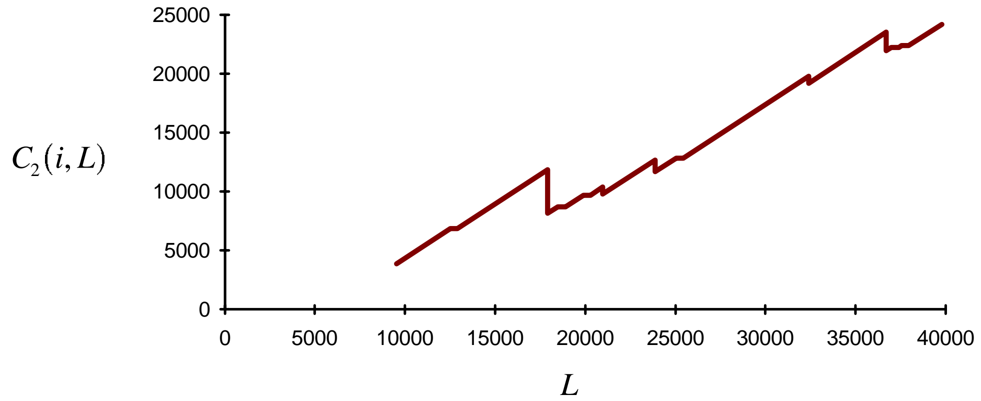

Figure 4-15: Graph of Condition 2 for Transmit Complete Task $(9,545 < L < 39,773)$

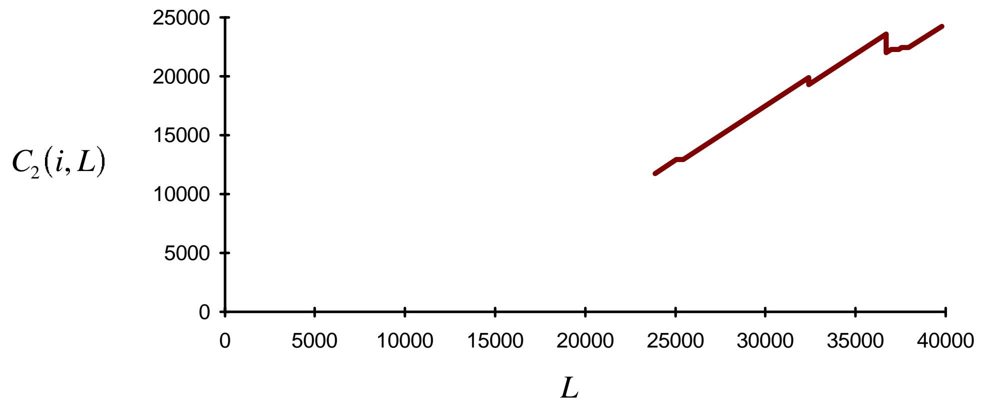

Figure 4-16: Graph of Condition 2 for User Tick Task $(23,864 < L < 39,773)$

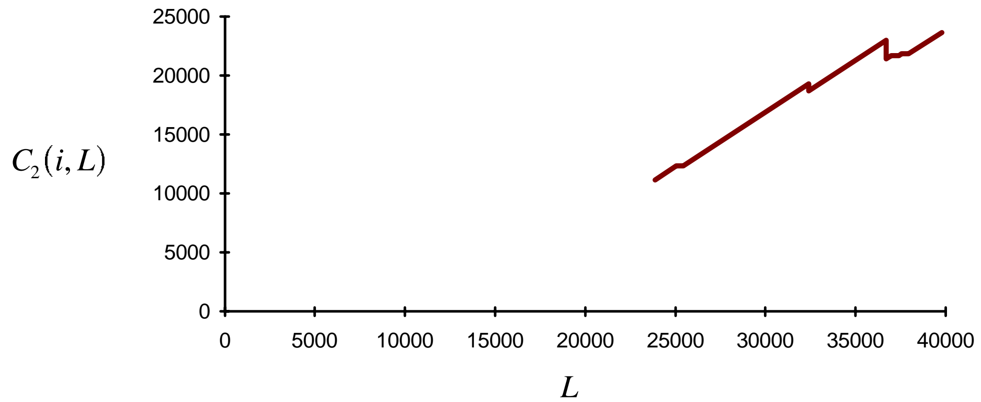

Figure 4-17: Graph of Condition 2 for Keyboard Check Task $(23,864 < L < 39,773)$

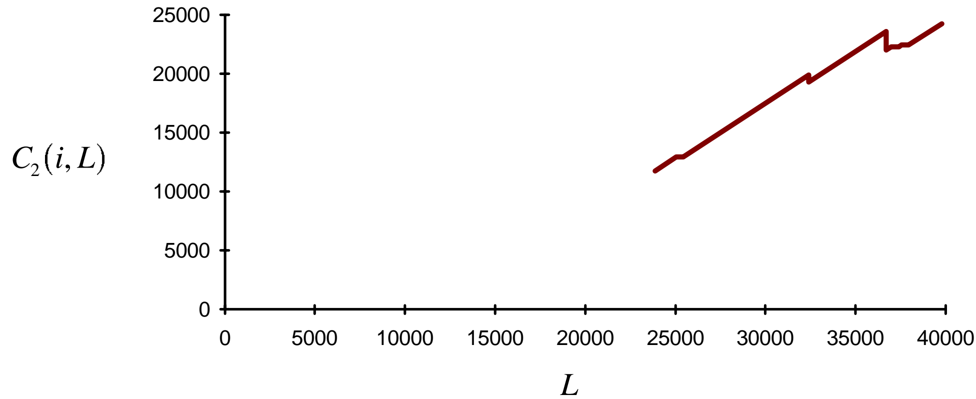

Figure 4-18: Graph of Condition 2 for Screen Output Task ($23,864 < L < 39,773$)

Thus, Conditions 1 and 2 hold for the task system τ and thus the task system is feasible. From this, I conclude that invocations of the application tasks comprising the acquisition-side of the application always execute to completion prior to their deadlines and that the tasks that share resources adhere to the required mutual exclusion constraints.

4.6 Summary

In this chapter, I have developed a formal model of acquisition-side of the workstation-based videoconferencing application. The key problem that was addressed in developing this formal model was that of determining the minimum interarrival time of each interrupt handler and application task. To solve this problem, I developed a set of techniques for determining the minimum interarrival time of interrupts and a set of rules for determining the minimum interarrival time of application tasks that are invoked by messages sent from interrupt handlers. Finally, I applied the feasibility test developed in Chapter 3 to the formal model in order to demonstrate that the application tasks comprising the acquisition-side of the application always execute to completion prior to their deadlines and that the tasks that share resources adhere to the required mutual exclusion constraints.

Chapter V
Analysis of the Delay Bound

5.1 Introduction

In Chapter 3 I developed a feasibility test for an abstract model of real-time systems that matched the programming model of YARTOS. In Chapter 4, I used this test to show that the application tasks comprising the workstation-based video conferencing application described in Chapter 2 always execute to completion prior to their deadlines and that the tasks that share resources adhere to the required mutual exclusion constraints. In this chapter, I present an axiomatic specification of the application in which these properties are included as axioms. From this specification, I derive a theorem showing that every video frame generated by the ActionMedia hardware is acquired and compressed, and that the delay experienced by video frames during processing on the acquisition-side is bounded.

The *acquisition-side delay* experienced by a video frame is precisely defined as the interval between two events: the VBI logical interrupt that occurs at the start of digitization of the frame and the time the frame is placed on the `transmit_video` queue (see Section 2.5.3). In this chapter, I will demonstrate that 100 ms. (*i.e.*, 6 times the period of the VBI logical interrupt) is an upper bound on the acquisition-side side delay experienced by each video frame.

Throughout the axiomatic specification of the application presented in this chapter, I take advantage of the deadline and mutual exclusion properties that were shown in Chapter 4. These properties are used both explicitly and implicitly. Explicitly, the fact that each task invocation completes execution prior to a deadline is used to show that each task invocation executes within a well-defined interval. In addition, the mutual exclusion property is used to show that several pairs of operations execute in mutual exclusion.

These assumptions are also used implicitly. The fact that tasks that share resources execute in mutual exclusion ensures that the effect of a task invocation can be modeled in

isolation, without interference from other tasks. In addition, the model contains few assumptions about the order in which task invocations execute: the only assumptions are that task invocations complete prior to their deadline and that tasks that share resources execute in mutual exclusion.

It should be noted that the model presented in this chapter is not complete, in the sense that the axioms presented do not represent all aspects of the behavior of the system. Rather, only the axioms that are necessary to show the desired properties are presented. As an example, it will be shown that, in most cases, the state conditions used in conditional statements will always hold when the conditional statement executes. Thus, for most conditional statements, axioms will be included in the model that represent the effect of executing the body of the conditional statement, while axioms representing the effect of not executing the body of the statement are omitted.

The axiomatic specification of the application presented in this chapter uses a formal language developed by Jahanian and Mok called RTL (Real-Time Logic) [22]. In Section 5.2, I describe the RTL notation. In Section 5.3, I define the basic concepts that will be used to develop an RTL model of the application. In Section 5.4, I describe the properties of the application that I would like to show, frame these properties as a correctness condition, and present the RTL expression representing this condition. In Section 5.5, I present RTL axioms that represent a number of basic properties of the application. In Section 5.6, I present the axioms that formalize the descriptions of the application tasks given in Section 2.5.9. Finally, in Section 5.7, I develop the proof of the correctness condition from the axiomatic specification presented earlier in the chapter.

5.2 Overview of Real-Time Logic

RTL is a formal language used to reason about occurrences of events. In this section, I present a subset of RTL sufficient to reason about the events that occur during execution of the workstation-based video conferencing application and to specify the properties of the application I wish to demonstrate.

RTL is a first-order logic. As such, formulas of RTL are formed from constants, variables, functions, predicates, universal and existential quantifiers and first-order logical connectives. There are three types of constants in RTL: integers, actions, and events. Variables range over integer, action and event constants. Functions include standard integer arithmetic functions (*i.e.*, addition, subtraction, etc.), and the *occurrence function*

(explained below). Predicates include standard integer comparison predicates (*i.e.*, equality, less than, etc.).

Action constants in an RTL specification of a system represent operations, or groups of operations, that are performed during execution of the system. For example, an action constant might represent the execution of a particular assignment statement or the execution of a subroutine. Actions can also be composite, in the sense that one action may be performed as part of another action. In the specification of the application given later in the chapter, the set of action constants includes constants representing both the execution of primitive operations (*e.g.*, the execution of a `digitize` operation) and the execution of entire application tasks (*e.g.*, the execution of the `VBI` task).

Event constants in RTL can be divided into three types: external events, start events, and stop events. External events model events generated by processes external to the system being specified (*i.e.*, an interrupt). An external event constant is denoted ΩE where E is the name of the external event. Start and stop events model the events corresponding to the initiation and completion of actions. A start event for an action A is denoted $\uparrow A$ and a stop event for the action is denoted $\downarrow A$.

Time is included in RTL by means of the *occurrence function*. Time is represented by positive integers (throughout the specification in this chapter, time will be represented in ticks as defined in Chapter 4). The occurrence function is a mapping from an event and a positive integer i to a positive integer representing the time of the ith occurrence of the event. The occurrence function is denoted $@(E,i)$.

As an example of an RTL specification, consider a system that includes a timer interrupt generated every 10 ticks, and a task that executes in response to the timer interrupt. Assume the task is guaranteed to complete within 6 ticks. Let $\Omega TIMER$ be the event corresponding to the timer interrupt, and let $TASK$ be the action corresponding to the execution of the task. Thus, $\uparrow TASK$ and $\downarrow TASK$ are the start and stop events corresponding to the start and completion of the task. The timer interrupt can be modeled with the axiom

```
@(ΩTIMER,i) = 10i
```

That is, the ith timer interrupt occurs at time $10i$. The execution of the task can be modeled with the axiom

```
   @(↑TASK,i) ≥ @(ΩTIMER,i)
 ∧ @(↓TASK,i) ≤ @(ΩTIMER,i) + 6
```

That is, the ith invocation of *TASK* starts execution sometime at or after the ith timer interrupt and completes execution within 6 ticks after the ith timer interrupt.

The above description of the RTL subset that will be used in the remainder of the chapter has left out several concepts present in the original definition of RTL given by Jahanian and Mok in [22]. In full RTL, a fourth kind of event constant and another kind of predicate are included. *Transition events* and *state predicates* are used to model assertions about the state of a system during an interval. For my purposes, these concepts are not necessary and thus have been omitted.

5.3 Basic Concepts

5.3.1 Symbolic Constants

I begin the specification with a discussion of several symbolic constants that will be used throughout the remainder of the chapter. These symbolic constants are listed in Figure 5-1 and represent several values, such as task minimum interarrival times and task deadlines, that were specified in the task declarations given in Section 2.5.9 or determined in Section 4.5.2.

Name	Value	Explanation
dvi_delay	1,098	Max. completion time of DVI handler
vbi_period	19,886	Period of VBI logical interrupt
vbi_deadline	17,898	Relative deadline of VBI task
vbi0_deadline	17,898	Relative deadline of VBI0 task
vbi1_deadline	17,898	Relative deadline of VBI1 task
cc_deadline	9,545	Relative deadline of CC task
compress_request	33,409	Max. time to complete compression operation
digitize_buffers	3	Max. buffers to hold digitized video frames
compress_buffers	10	Max. buffers to hold compressed frames
max_transport	8	Max. buffers in transport system

Figure 5-1: Symbolic Constants

I choose to represent these values symbolically both to clarify the presentation and to decouple the analysis presented here from the specific deadlines, etc., chosen in earlier chapters. However, what will be required here is that certain relationships between the values of the symbolic constants hold. These are listed in Figure 5-2.

$$digitize_buffers \geq 3$$
$$compress_buffers \geq max_transport + 2$$
$$compress_request \leq 2 \cdot vbi_period$$
$$dvi_delay + vbi_deadline \leq vbi_period$$
$$dvi_delay + vbi0_deadline \leq vbi_period$$
$$dvi_delay + vbi1_deadline \leq vbi_period$$
$$dvi_delay + cc_deadline \leq vbi_period$$

Figure 5-2: Relationships Among Symbolic Constants

5.3.2 Action Constants

There are six groups of action constants in the RTL model of the application: task actions, subtask actions, message actions, queuing actions, memory management actions, and video frame processing actions. The task actions represent the execution of application tasks. The first four represent the tasks that are involved in the acquisition and compression of video frames: `VBI`, `VBI0`, `VBI1`, and `CC` (see Section 2.5). The action names used to represent these tasks are listed in Figure 5-3.

Task	Action
VBI	vbi_task
VBI0	vbi0_task
VBI1	vbi1_task
CC	cc_task
transmit_complete	tc_task

Figure 5-3: Task Actions

Subtask actions represent the execution of a group of statements within an application task. In the RTL model of the application, only the `VBI1` task is assumed to contain subtasks. This task has two subtasks defined by the two “if” statements (see Figure 2-22). The action names used to represent these subtasks are listed in Figure 5-3.

Task	Subtask 1	Subtask 2
vbi1_task	vbi1_part1	vbi1_part2

Figure 5-3: Subtask Actions

The message actions represent the execution of `send_message` system calls (see Section 2.4.2). Recall that each time an application task receives a message, an invocation of the task is created and assigned a logical arrival time (see Section 2.4.5). In the model, both the actual arrival time and the logical arrival time of each message must be

represented. Thus, for each `send_message` call in the application, two actions are included in the model: a *send action*, and a *logical send action*. The send action represents the actual execution of the `send_message` call. The logical send action is artificial; it is assumed to have occurred at the logical arrival time of the message. Figure 5-5 lists the message actions.

Receiving Task	Send Action	Logical Send Action
vbi_task	send_vbi	logical_send_vbi
vbi0_task	send_vbi0	logical_send_vbi0
vbi1_task	send_vbi1	logical_send_vbi1
cc_task	send_cc	logical_send_cc

Figure 5-5: Message Actions

The queuing actions represent the execution of `insert_queue` and `remove_queue` operations (see Section 2.5.1). For each queue in the application that holds digitize buffers or compress buffers (*i.e.*, a video frame), two actions are included in the model: one that represents inserting a buffer on the queue, and one that represents removing a buffer from the queue. The queuing actions are listed in Figure 5-6.

Queue	Insert Queue Action	Remove Queue Action
next_digitize	put_next_digitize	get_next_digitize
digitizing	put_digitizing	get_digitizing
compress_source	put_compress_source	get_compress_source
compress_sink	put_compress_sink	get_compress_sink
video_transmit	put_transmit	get_transmit

Figure 5-6: Queuing Actions

The memory management actions represent the execution of `allocate` and `free` operations for buffers used to hold video frames (see Section 2.5.1). For both digitize buffers and compress buffers, two actions are included in the model: one representing an `allocate` operation for that kind of buffer, and one representing a `free` operation for that kind of buffer. These actions are listed in Figure 5-7.

Data Type	Allocate Action	Free Action
digitize buffers	alloc_digitize	free_digitize
compress buffers	alloc_compress	free_compress

Figure 5-7: Memory Management Actions

Video frame processing actions represent the execution of `digitize` and `start_compress` operations (see Section 2.5.1). These operations initiate the digitization and compression of video frames by the ActionMedia hardware. These actions are listed in Figure 5-8.

Operation	Video Frame Processing Action
digitize	digitize
start_compress	compress

Figure 5-8: Video Frame Processing Actions

5.3.3 Event Constants

The RTL model of the application includes a number of event constants. First, the model includes a start event and a stop event for each of the actions listed above. In addition, the model includes two external events corresponding to the VBI logical interrupt and the CC logical interrupt. These external events are listed in Figure 5-9.

Logical Interrupt	External Event
VBI	ΩVBI
CC	ΩCC

Figure 5-9: External Events

5.3.4 Frame Numbers

As discussed in Chapter 2, a video frame is acquired and digitized by the ActionMedia hardware over an interval of approximately 33 ms. In particular, a new frame is acquired and digitized over an interval between two even-numbered VBI logical interrupts. Throughout the remainder of the chapter, I will refer to individual video frames using a *frame number* that is defined based on the index of the VBI interrupt corresponding to the start of the interval in which the frame was acquired. Specifically, frame number i refers to the video frame acquired between times @(ΩVBI,2i) and @(ΩVBI,2i+2).

With the occurrence function, I am able to reason about the time at which operations execute; for example I can compare the time of the ith `digitize` operation to the time of the jth `compress_start` operation. However, I will often wish to reason about the relationship between several operations performed on a particular frame. In order to capture the correspondence between frame numbers and the operations performed on those frames, I define a pair of mappings:

```
frame: (action, i) → frame number

index: (action, frame number) → i
```

For each queuing action and video frame processing action, *frame* maps the action and a positive integer *i* to the frame number of the frame operated on by the ith occurrence of the action. It is often used in expressions of the following form:

```
  frame(action1, i) = n
⇒
  frame(action2, i) = n
```

That is, if the ith occurrence of `action1` operated on frame *n*, then the ith occurrence of `action2` also operated on frame *n*.

Index is the inverse mapping of *frame*. For each queuing action and video frame processing action, it maps the action and a frame number *j* to *i* if and only if the ith occurrence of the action was performed on the frame with frame number *j*. This mapping is often used in expressions of the following form:

```
@(↓action,index(action,n)) < t
```

That is, the time at which `action` was completed on the frame with frame number *n* was less than *t*.

5.4 Correctness Conditions

In this section, I give an RTL specification of the conditions that must hold for a video frame to be correctly acquired, compressed, and readied for transmission over the network (see Section 2.5.3). Most of these conditions are constraints on the timing and ordering of operations (*e.g.*, the compression of a frame cannot begin until digitization is complete). If these conditions do not hold, then frames will be discarded or corrupted. In addition, I add one more correctness constraint: the processing delay incurred by the video frame must be bounded.

As described in Section 2.5.3, the ActionMedia hardware continuously acquires and digitizes video frames, and writes the digitized data into a digitize buffer specified by the application. The application specifies a new buffer by executing a `digitize` operation in response to an odd-numbered VBI interrupt. At the next VBI interrupt, the hardware begins writing into the specified buffer. For the video frame to be acquired correctly, the

ActionMedia hardware must continue to write data to the buffer until the next even-numbered VBI interrupt. For frame *i*, this is the interval between @(ΩVBI,2i) and @(Ω VBI,2i+2).

Thus, in order to ensure that frame *i* is acquired correctly, the application must execute two `digitize` operations. The first must be executed in the interval between @(Ω VBI,2i-1) and @(ΩVBI,2i) and must specify the digitize buffer into which frame *i* is to be written. If the `digitize` operation is executed earlier, then data that is not part of frame *i* will be written to the buffer; if it is executed later, then some of the data for frame *i* will be written to a different buffer (*i.e.*, the buffer that was passed to the ActionMedia hardware by the previous `digitize` operation). Expressed in RTL, this condition is

```
  @(↑digitize,index(digitize,i)) > @(ΩVBI,2i-1)
∧ @(↓digitize,index(digitize,i)) ≤ @(ΩVBI,2i)
```

The second `digitize` operation that must be executed in order to ensure that frame *i* is correctly acquired must occur in the interval between @(ΩVBI,2i+1) and @(ΩVBI,2i+2) and must specify a new buffer to hold frame *i*+1. If this `digitize` operation is executed any earlier, then some of the data for frame *i* will be written to the new buffer; if it executes any later, then some of the data for frame *i* will be overwritten by data from frame *i*+1. Expressed in RTL, this condition is

```
∃j [   @(↑digitize,j) > @(ΩVBI,2i+1)
     ∧ @(↓digitize,j) ≤ @(ΩVBI,2i+2)]
```

Finally, the application may not execute any other `digitize` operations in the interval between @(ΩVBI,2i-1) and @(ΩVBI,2i+1). If it did, then some of the data for frame *i* would be written to the newly specified buffer. Expressed in RTL, this condition is

```
~∃j [   j≠index(digitize,i)
      ∧ @(↑digitize,j) > @(ΩVBI,2i-1)
      ∧ @(↓digitize,j) ≤ @(ΩVBI,2i+1)
     ]
```

For a frame to be compressed correctly, we need to ensure that the `start_compress` operation does not occur until the digitization is complete. For frame *i*, this condition is expressed in RTL as

```
@(↑compress,index(compress,i)) ≥ @(ΩVBI,2i+2)
```

Next, a frame should not be placed on the `transmit_video` queue until compression has finished. For frame *i*, this condition is expressed in RTL as

```
@(↑put_transmit,index(put_transmit,i)) ≥ @(ΩCC,index(compress,i))
```

Finally, the acquisition-side delay of a video frame should be bounded. Recall that the acquisition-side delay is defined as the length of the interval between two events: the VBI logical interrupt that occurs at the start of digitization of the frame and the time the frame is placed on the `transmit` queue. In the remainder of the chapter, I will show that this condition can be shown for a bound of 6·*vbi_period*. For frame *i*, this condition is expressed in RTL as

```
@(↓put_transmit,index(put_transmit,i)) - @(ΩVBI,2i) ≤ 6·vbi_period
```

Altogether, an RTL specification of the conditions that must hold for video frame *i* to be correctly acquired, compressed, and readied for transmission, with bounded delay, is the conjunction of the above conditions. The full correctness condition is listed in Figure 5-10.

```
  @(↑digitize,index(digitize,i)) > @(ΩVBI,2i-1)
∧ @(↓digitize,index(digitize,i)) ≤ @(ΩVBI,2i)
∧  ∃j [  @(↑digitize,j) > @(ΩVBI,2i+1)
       ∧ @(↓digitize,j) ≤ @(ΩVBI,2i+2)]
∧ ~∃j [  j≠index(digitize,i)
       ∧ @(↑digitize,j) > @(ΩVBI,2i-1)
       ∧ @(↓digitize,j) ≤ @(ΩVBI,2i+1)
      ]
∧ @(↑compress,index(compress,i)) ≥ @(ΩVBI,2i+2)
∧ @(↑put_transmit,index(put_transmit,i)) ≥ @(ΩCC,index(compress,i))
∧ @(↓put_transmit,index(put_transmit,i)) - @(ΩVBI,2i) ≤ 6·vbi_period
```

Figure 5-10: Correctness Condition for a Video Frame

5.5 Basic Axioms and Theorems

In this section, I begin presenting the axioms that model the behavior of the application. In a number of cases, the model of the application includes a set of axioms that have the same form, but are defined for different actions. For example, corresponding to each action in the model, there is an axiom that represents the fact that the action starts before it completes. For the `digitize` action, this axiom is

```
@(↑digitize,i) < @(↓digitize,i)
```

To simply the presentation, I will present the set of axioms of this form as a single "generic" axiom. That is, in the description below, I present the following axiom and specify that it is defined for each of the actions in Figures 5-3 through 5-8:

```
@(↑action,i) < @(↓action,i)
```

The interpretation of this is that a set of axioms should be included in the model, with the bold-faced name **action** instantiated by each specified action.

In addition to the axioms presented in this section, I derive several theorems that will be used throughout the remainder of the chapter. These theorems are also presented as "generic theorems" that are instantiated for a number of actions.

5.5.1 Actions

The first two generic axioms in the RTL model of the application represent two simple constraints on the execution of each action. Axiom 5.1 represents the fact that an action starts before it completes. Axiom 5.2 represents the fact that the i+1st occurrence of an action cannot start until the ith occurrence of the action completes. Thus for each of the actions in Figures 5-3 through 5-8, axioms of the following form are included in the model:

Axiom 5.1

```
@(↑action,i) < @(↓action,i)
```

Axiom 5.2

```
@(↓action,i) ≤ @(↑action,i+1)
```

From these axioms, I now derive a simple theorem that applies to all actions. Theorem 5.3 shows that for an action A and all i less than or equal to j, the ith instance of A begins execution at or before the jth instance of A begins execution, and completes execution at or before the jth instance of A completes execution.

Theorem 5.3

For each action in Figures 5-3 through 5-8, and for all $i \leq j$

```
  @(↑action,i) ≤ @(↑action,j)
∧ @(↓action,i) ≤ @(↓action,j)
```

Proof: By induction on *j*.

Base case: Assume $j = i$. The theorem holds trivially.

Inductive case: Assume that the theorem holds for $j \leq k$. By the inductive assumption, Axiom 5.1, and Axiom 5.2

```
@(↑action,i) ≤ @(↑action,k)
             ≤ @(↓action,k)
             ≤ @(↑action,k+1)                    (5.1)
```

Similarly, by the inductive assumption, Axiom 5.2, and Axiom 5.1

```
@(↓action,i) ≤ @(↓action,k)
             ≤ @(↑action,k+1)
             ≤ @(↓action,k+1)                    (5.2)
```

Combining (5.1) and (5.2)

```
  @(↑action,i) ≤ @(↑action,k+1)
∧ @(↓action,i) ≤ @(↓action,k+1)
```

This proves the theorem. □

5.5.2 Frame Numbers

The next group of axioms deal with frame numbers and the rules used to associate frame numbers with operations. The first axiom establishes a correspondence between `digitize` operations, frame numbers, and VBI logical interrupts. Recall that a frame is acquired by executing a `digitize` operation in response to an odd-numbered VBI interrupt, and that the ActionMedia hardware begins writing a digitized frame to the specified starting at the next VBI interrupt. Frame *i* is defined as the frame that is written starting at time @(ΩVBI,2i). Thus, if the jth `digitize` completed in the interval @(Ω VBI,2i-1) to @(ΩVBI,2i), then the frame acquired in response to the jth `digitize` operation must be frame *i*. This is the rule represented by Axiom 5.4.

Axiom 5.4

```
[  @(↓digitize,j) > @(ΩVBI,2i-1)
 ∧ @(↓digitize,j) ≤ @(ΩVBI,2i)]
⇒
  frame(digitize,j) = i
```

The next axiom establishes the FIFO property of the queues used in the application. Since queues are FIFO, the ith `remove_queue` operation on a queue retrieves the data put into the queue by the ith `insert_queue` operation. For each pair of queuing actions listed in Figure 5-6, this property is included in the model with an axiom of the following form:

Axiom 5.5

```
frame(put_queue,i) = frame(get_queue,i)
```

Finally, recall that the *index* mapping is the inverse mapping of *frame*. To represent this property, an axiom of the following form is included in the model for each queuing action in Figure 5-6 and video frame processing action in Figure 5-8:

Axiom 5.6

```
frame(action,i) = j ⇔ index(action,j) = i
```

5.5.3 Hardware Interrupts

The next group of axioms models the behavior of the hardware interrupts and interrupt handlers involved in the acquisition and compression of video frames. These interrupts are the VBI logical interrupt and the CC logical interrupt, which are represented in the model by the external events ΩVBI and ΩCC. The VBI logical interrupt is periodic; I will assume that the first interrupt occurs at time 0, and successive interrupts occur periodically every *vbi_period* time units. Thus, the behavior of the VBI interrupt is captured by the axiom:

Axiom 5.7

```
@(ΩVBI,i) = (i-1)·vbi_period
```

The CC logical interrupt occurs when the compression of a video frame is finished. The compression of a frame is initiated by the `compress` action and is assumed to finish within an interval defined by the symbolic constant *compress_request* (see Figure 5-1). Thus, the behavior of the CC logical interrupt is captured by the axiom:

Axiom 5.8

```
  @(ΩCC,i) ≥ @(↓compress,i)
∧ @(ΩCC,i) ≤ @(↓compress,i) + compress_request
```

Both the VBI and CC logical interrupts are handled by the DVI interrupt handler. This handler determines which logical interrupt has occurred and then sends a message to the appropriate task (either the `VBI` or the `CC` task). As shown by the analysis performed in Chapter 4, the DVI interrupt handler completes executes within an interval defined by the symbolic constant *dvi_delay* (see Figure 5-1). Thus, if the DVI interrupt handler sends a message, it will be sent within *dvi_delay* ticks after the interrupt. Furthermore, the logical arrival time assigned to the receiving task will also be within *dvi_delay* ticks after the interrupt (see Section 2.4.5).

Axiom 5.9 represents the execution of the DVI interrupt handler in response to a VBI logical interrupt. The interpretation of this axiom is that two operations, a send action and a logical send action, occur in the interval between the interrupt and the upper bound on the time it must complete, [@(ΩVBI,i), @(ΩVBI,i) + *dvi_delay*].

Axiom 5.9

```
  @(↑send_vbi,i) ≥ @(ΩVBI,i)
∧ @(↓send_vbi,i) ≤ @(ΩVBI,i) + dvi_delay
∧ @(↑logical_send_vbi,i) ≥ @(ΩVBI,i)
∧ @(↓logical_send_vbi,i) ≤ @(ΩVBI,i) + dvi_delay
```

Axiom 5.10 represents the execution of the DVI interrupt handler in response to a CC logical interrupt. Again, the interpretation of this axiom is that two operations, a send action and a logical send action, occur in the interval between the interrupt and the upper bound on the time it must complete, [@(ΩCC,i), @(ΩCC,i) + *dvi_delay*].

Axiom 5.10

```
  @(↑send_cc,i) ≥ @(ΩCC,i)
∧ @(↓send_cc,i) ≤ @(ΩCC,i) + dvi_delay
∧ @(↑logical_send_cc,i) ≥ @(ΩCC,i)
∧ @(↓logical_send_cc,i) ≤ @(ΩCC,i) + dvi_delay
```

5.5.4 Task Scheduling and Execution

The next group of axioms represent constraints on the execution of task invocations. Recall that each task invocation executes in response to a message. Furthermore, as a result of the analysis performed in Chapter 4, we know that each task invocation will complete execution prior to its deadline. Thus, each time a task receives a message, the task invocation will begin execution after it receives a message, and complete execution prior to its logical arrival time plus its relative deadline. This property is represented by Axiom 5.11. An instance of this axiom is defined for each triplet of receiving task, send

action, and logical send action listed in Figure 5-5. The bold-faced symbol **task_deadline** should be instantiated with the relative deadline of the receiving task defined in Figure 5-1.

Axiom 5.11

```
  @(↑receiving_task,i) ≥ @(↓send_action,i)
∧ @(↓receiving_task,i) < @(↓logical_send_action,i) + task_deadline
```

5.5.5 Subtask Execution

The next axiom represents constraints on the execution of the subtasks of the `VBI1` task. Recall that the `VBI1` task has two subtasks defined by the two "if" statements in its body (see Figure 2-22). Each time an invocation of the VBI1 task executes, the two subtasks execute in order. This property is represented by the following axiom:

Axiom 5.12

```
  @(↑vbi1_task,i) < @(↑vbi1_part1,i)
∧ @(↓vbi1_part1,i) < @(↑vbi1_part2,i)
∧ @(↓vbi1_part2,i) < @(↓vbi1_task,i)
```

5.5.6 Mutual Exclusion

When two tasks share a resource, those tasks are guaranteed not to preempt one another. Thus, invocations of one task do not overlap with invocations of another task with which it shares a resource. Thus, if **task1** and **task2** are tasks that share a resource, then the following property can be asserted about the relationship between invocations of the tasks: if the ith instance of **task1** started execution before the jth instance of **task2** started execution, then it must also have completed execution before the jth instance of **task2** started execution. This property can be included in the model with an axiom of the following form:

Axiom 5.13

```
  @(↑task1,i) < @(↑task2,j)
⇒
  @(↓task1,i) < @(↑task2,j)
```

Figure 5-11 lists pairs of tasks for which Axiom 5.13 is defined.

vbi0_task	cc_task
tc_task	cc_task

Figure 5-11: Actions Performed in Mutual Exclusion

5.5.7 At-Most-Once Actions

The next group of axioms establishes some particularly useful properties of a set of actions referred to as *at-most-once* actions. An at-most-once action is an action that is performed in only one task (or subtask), and is executed at most once during a single invocation of that task. Figure 5-12 lists each at-most-once action along with the task (or subtask) that executes that action.

AMO Action	Task
send_vbi0	vbi_task
logical_send_vbi0	vbi_task
send_vbi1	vbi_task
logical_send_vbi1	vbi_task
put_next_digitize	vbi1_part2
get_next_digitize	vbi0_task
put_digitizing	vbi0_task
get_digitizing	vbi0_task
put_compress_source	vbi0_task

AMO Action	Task
get_compress_source	vbi1_part1
put_compress_sink	vbi0_task
get_compress_sink	cc_task
put_transmit	cc_task
alloc_digitize	vbi1_part2
free_digitize	vbi1_part1
alloc_compress	vbi0_task
digitize	vbi1_part2
compress	vbi0_task

Figure 5-12: At-Most-Once Actions

This property can be included in the RTL model with three generic axioms defined for each pair of AMO actions and tasks in Figure 5-12. Axiom 5.14 represents the fact that the ith at-most-once action performed by a task cannot begin until the start of the ith invocation of the task.

Axiom 5.14

```
@(↑amo_action,i) > @(↑task,i)
```

Axiom 5.15 represents the fact that if the ith at-most-once action performed by a task starts after the jth invocation of the task completes, then it must actually start after the $j+1^{st}$ invocation of the task begins execution.

Axiom 5.15

```
  @(↑amo_action,i) ≥ @(↓task,j)
⇒
  @(↑amo_action,i) > @(↑task,j+1)
```

Finally, Axiom 5.16 represents the fact that if the ith at-most-once action starts after the jth invocation of the task, then the $i+k^{th}$ action cannot be performed until at least the start of the $j+k^{th}$ invocation of the task.

Axiom 5.16

```
   @(↑amo_action,i) > @(↑task,j)
 ⇒
   @(↑amo_action,i+k) > @(↑task,j+k)
```

I now present a pair of simple and useful theorems for at-most-once actions. Theorem 5.17 shows that if the first at-most-once action *A* executed by a task *T* begins execution after the jth invocation of *T* begins execution, and there is some *k* such that the kth instance of *A* is performed by the jth invocation of *T*, then *k* must be one. Theorem 5.18 is similar; if the i-1st instance of *A* is performed by the j-1st invocation of *T*, and if there is some *k* such that the kth instance of *A* is performed by the jth invocation of *T*, then *k* must be *i*.

Theorem 5.17

For each pair of AMO actions and tasks in Figure 5-12,

```
     @(↑amo_action,1) > @(↑task,j)
   ∧ @(↑amo_action,k) > @(↑task,j)
   ∧ @(↓amo_action,k) < @(↓task,j)
 ⇒
   k = 1
```

Proof: Assume the l.h.s. of the implication. Since the second argument of the occurrence function is defined to be a positive integer, it is the case that $k \geq 1$. I now show that $k = 1$ by contradiction. Assume $k \geq 2$. By Theorem 5.3

```
@(↑amo_action,k) ≥ @(↑amo_action,2)
```

By the l.h.s. of the theorem and Axiom 5.16, for all *n*

```
@(↑amo_action,1+n) > @(↑task,j+n)
```

Combining these facts yields equation (5.3).

```
@(↑amo_action,k) ≥ @(↑amo_action,2)
                 > @(↑task,j+1)                    (5.3)
```

However by Axiom 5.1, the l.h.s. of the theorem, and Axiom 5.2

```
@(↑amo_action,k) < @(↓amo_action,k)
                 < @(↓task,j)
                 < @(↑task,j+1)
```

which contradicts (5.3). Thus, $k < 2$. This proves the theorem. □

Theorem 5.18

For each pair of AMO actions and tasks in Figure 5-12, and for $i > 1$

```
    @(↑amo_action,i-1) > @(↑task,j)
  ∧ @(↓amo_action,i-1) < @(↓task,j)
  ∧ @(↑amo_action,k) > @(↑task,j+1)
  ∧ @(↓amo_action,k) < @(↓task,j+1)
⇒
  k = i
```

Proof: Assume the l.h.s. of the implication. By this assumption, and Axioms 5.2 and 5.1

```
@(↑amo_action,k) > @(↑task,j+1)
                 > @(↓task,j)
                 > @(↓amo_action,i-1)
                 > @(↑amo_action,i-1)
```

Thus, by the contrapositive of Theorem 5.3, $k > i-1$ and thus $k \geq i$.

I now show that $k = i$ by contradiction. Assume $k \geq i+1$. Then by Theorem 5.3

```
@(↑amo_action,k) ≥ @(↑amo_action,i+1)
```

By the l.h.s. of the theorem and Axiom 5.16, for all n

```
@(↑amo_action,i-1+n) > @(↑task,j+n)
```

Combining these facts yields equation (5.4)

```
@(↑amo_action,k) ≥ @(↑amo_action,i+1)
                 > @(↑task,j+2)                                  (5.4)
```

However, by Axiom 5.1, the l.h.s. of the theorem, and Axiom 5.2

```
@(↑amo_action,k)< @(↓amo_action,k) < @(↓task,j+1) < @(↑task,j+2)
```

which contradicts (5.4). Thus, $k < i+1$. This proves the theorem. □

5.6 Task Descriptions

5.6.1 Representing Conditional Statements in RTL

In this section, I present the axioms the represent the effect of executing application tasks. I begin with a discussion of the technique used to create an RTL representation of the

conditional statements used in the application tasks. Consider a conditional statement executed by the ith invocation of a task.

```
if (condition) then
   action
end if
```

If the condition is constrained such that its value cannot change between the start of the task and the execution of the test, then this statement can be represented in RTL with the following assertion (assuming **condition** is an RTL representation of an assertion that the condition holds at the start of the task execution):

```
   condition
 ⇒
   ∃j [  @(↑action,j) > @(↑task,i)
       ∧ @(↓action,j) < @(↓task,i)]
```

That is, a conditional statement is represented as an implication; the left-hand side is an RTL representation of the condition and the right-hand side is an expression representing the execution of the action by the task. Specifically, the expression on the right-hand side can be interpreted as an assertion that for some j, the jth instance of **action** started execution after the ith invocation of **task** started execution and completed before the ith invocation of **task** finished execution.

Conditional statements within subtasks can be represented using the same technique. That is, if the condition in a conditional statement is constrained such that its value cannot change between the start of the subtask and the execution of the test, then the statement can be represented using an assertion of the form given above, with **subtask** in place of **task**.

The tasks and subtasks specified in this chapter use a number of conditional statements that are based on state conditions. One common state condition is that the length of a queue is greater than zero. If the queue is declared as a resource by each task that accesses it, and any changes to the queue by the task (or subtask) in question occur after the test of the state condition, then the value of the state condition will not change between the time an invocation of the task starts execution and the time it executes the conditional statement. An RTL expression that represents **queue** having non-zero length when the ith invocation of **task** begins execution is:

```
∃j [  @(↓put_queue,j) ≤ @(↑task,i)
    ∧ @(↑get_queue,j) > @(↑task,i)]
```

That is, the length of the queue is greater than zero at the time the ith invocation of **task** begins execution if there is some j for which the jth `insert_queue` operation on the queue has already occurred, and the jth `remove_queue` on the queue has not yet occurred.

A state condition similar to the non-zero length of a queue is the non-empty state of a pool of free buffers (as tested by the `available` operation). Again, if the pool of free buffers is a resource, and the task (or subtask) in question does not allocate or free buffers before the test of the state condition, then the value of the state condition will not change between the time an invocation of the task starts execution and the time it executes the conditional statement. If it is assumed that **buffers** is total number of buffers of **type** in the system, then an RTL expression that represents a free buffer of **type** being available when the ith invocation of **task** begins execution is:

```
   @(↑alloc_type,buffers) > @(↑task,i)
∨
   ∃m [  @(↑alloc_type,m+buffers) > @(↑task,i)
       ∧ @(↓free_type,m) ≤ @(↑task,i)]
```

That is, a buffer is available to be allocated under one of two conditions: the number of buffers allocated prior to the start of the task is less than **buffers**, or the difference between the number of buffers that have been allocated prior to the start of the task and the number of buffers that have been free prior to the start of the task is less than **buffers**.

5.6.2 RTL Specification of the VBI Task

As shown in Figure 2-21, the VBI task alternates sending messages to the `VBI1` and the `VBI0` tasks. On odd-numbered executions, it sends a message to the `VBI1` task; thus, the ith message to the `VBI1` task is generated by the $2i$-1st invocation of the `VBI` task. On even-numbered executions it sends a message to the `VBI0` task; thus the ith message to the `VBI0` task is generated by the $2i^{th}$ invocation of the `VBI` task. One other issue must be addressed in the specification of the `VBI` task: when a task invocation sends a message to another task, the resulting task invocation is assigned a logical arrival time equal to the logical arrival time of the sender (see Section 2.4.5). These properties of the `VBI` task are represented in the model by a pair of Axioms, 5.19 and 5.20.

Axiom 5.19

```
  @(↑send_vbi1,i) > @(↑vbi_task,2i-1)
∧ @(↓send_vbi1,i) < @(↓vbi_task,2i-1)
∧ @(↓logical_send_vbi1,i) = @(↓logical_send_vbi,2i-1)
```

Axiom 5.20

```
  @(↑send_vbi0,i) > @(↑vbi_task,2i)
∧ @(↓send_vbi0,i) < @(↓vbi_task,2i)
∧ @(↓logical_send_vbi0,i) = @(↓logical_send_vbi,2i)
```

5.6.3 RTL Specification of the VBI0 Task

As shown in Figure 2-23, the `VBI0` task consists of two conditional statements. In the first conditional, several actions are taken if: 1) the `digitizing` queue is not empty, and 2) a free compress buffer can be allocated. If these conditions hold, then a buffer is removed from the `digitizing` queue and placed on the `compress_source` queue, a new compress buffer is allocated and placed on the `compress_sink` queue, and a `start_compress` operation is executed on the digitize buffer and the compress buffer.

Axiom 5.21 represents this conditional statement for the ith invocation of the `VBI0` task. The left-hand-side of the implication is the conjunction of the expressions described above for representing the state conditions on the `digitizing` queue and the pool of free compress buffers. The right-hand-side of the implication represents the fact that the set of operations performed in the conditional occur during the execution of the ith invocation of the `VBI0` task. In addition, the right-hand-side of the implication equates the frames numbers associated with each of the operations.

Axiom 5.21

```
    ∃j [  @(↓put_digitizing,j) ≤ @(↑vbi0_task,i)
        ∧ @(↑get_digitizing,j) > @(↑vbi0_task,i)]
   ∧
     [  @(↑alloc_compress,compress_buffers) > @(↑vbi0_task,i)
      ∨
        ∃m [  @(↑alloc_compress,m+compress_buffers) > @(↑vbi0_task,i)
            ∧ @(↓free_compress,m) ≤ @(↑vbi0_task,i)]
     ]
  ⇒
  ∃k∃f [ @(↑get_digitizing,k) > @(↑vbi0_task,i)
     ∧   @(↓get_digitizing,k) < @(↓vbi0_task,i)
     ∧   frame(get_digitizing,k) = f
     ∧   @(↑put_compress_source,k) > @(↑vbi0_task,i)
     ∧   @(↓put_compress_source,k) < @(↓vbi0_task,i)
     ∧   frame(put_compress_source,k) = f
     ∧   @(↑alloc_compress,k) > @(↑vbi0_task,i)
     ∧   @(↓alloc_compress,k) < @(↓vbi0_task,i)
     ∧   @(↑put_compress_sink,k) > @(↑vbi0_task,i)
     ∧   @(↓put_compress_sink,k) < @(↓vbi0_task,i)
     ∧   frame(put_compress_sink,k) = f
     ∧   @(↑compress,k) > @(↑vbi0_task,i)
     ∧   @(↓compress,k) < @(↓vbi0_task,i)
     ∧   frame(compress,k) = f]
```

In addition, it will be necessary to consider the case where the `digitizing` queue is empty when the `VBI0` task begins execution. In this case, the `get_digitizing` action and the `alloc_compress` action (among others) are not executed during the ith invocation of the `VBI0` task; in other words, all instances of these actions either complete prior to the start of the ith invocation of the `VBI0` task or begin after the completion of the ith invocation of the `VBI0` task. This fact is represented by the following axiom:

Axiom 5.22

```
    ~∃j [  @(↓put_digitizing,j) ≤ @(↑vbi0_task,i)
         ∧ @(↑get_digitizing,j) > @(↑vbi0_task,i)]
  ⇒
    ∀k [  @(↓get_digitizing,k) < @(↑vbi0_task,i)
        ∨ @(↑get_digitizing,k) > @(↓vbi0_task,i)]
      ∧
       [  @(↓alloc_compress,k) < @(↑vbi0_task,i)
        ∨ @(↑alloc_compress,k) > @(↓vbi0_task,i)]
```

In the second part of the `VBI0` task, a buffer is removed from the `next_digitizing` queue and inserted on the `digitizing` queue if the `next_digitizing` queue is not empty. Axiom 5.23 represents this conditional statement for the ith invocation of the

VBI0 task. The left-hand-side of the implication is an expression representing the state condition on the next_digitizing queue. The right-hand-side of the implication represents the fact that the set of operations performed in the conditional occur during the execution of the ith invocation of the VBI0 task. In addition, the right-hand-side of the implication associates the frame number of the frame removed from the next_digitizing queue with the put_digitizing action.

Axiom 5.23

```
    ∃j [ @(↓put_next_digitizing,j) ≤ @(↑vbi0_task,i)
       ∧ @(↑get_next_digitizing,j) > @(↑vbi0_task,i)]
  ⇒
    ∃k∃f [ @(↑get_next_digitizing,k) > @(↑vbi0_task,i)
         ∧ @(↓get_next_digitizing,k) < @(↓vbi0_task,i)
         ∧ frame(get_next_digitizing,k) = f
         ∧ @(↑put_digitizing,k) > @(↑vbi0_task,i)
         ∧ @(↓put_digitizing,k) < @(↓vbi0_task,i)
         ∧ frame(put_digitizing,k) = f]
```

5.6.4 RTL Specification of the VBI1 Task

As shown in Figure 2-22, the VBI1 task consists of two conditional statements. Since the test of the second conditional statement may depend on the execution of the first conditional statement, the VBI1 task is divided into the two subtasks listed in Figure 5-3.

In the first subtask, a digitize buffer is removed from the compress_source queue and freed if: 1) the compress_source queue is not empty, and 2) the compression of the first frame on the queue has completed. In the pseudo-code shown in Figure 2-22, the second test is performed by comparing the YARTOS eventcount of CC logical interrupts to db_freed, a variable that counts the number of digitize buffers removed from the compress_source queue and freed. This test will succeed if the number of get_compress_source actions occurring prior to the start of the subtask is less than the number of CC interrupts occurring prior to the start of the subtask, or equivalently, if there is some j such that the jth CC interrupt occurred prior to the start of the subtask and the jth get_compress_source occurs after the start of the subtask. This test can be represented in RTL by the following expression:

```
  ∃j [  @(↑get_compress_source,j) > @(↑vbi1_part1,i)
      ∧ @(ΩCC,j) ≤ @(↑vbi1_part1,i)]
```

Axiom 5.24 represents the effect of executing the first subtask during the ith invocation of the VBI1 task. The left-hand-side of the implication is the conjunction of the two conditions (combined into one) and the right-hand-side of the implication represents the fact that the set of operations performed in the conditional occur during the execution of the ith invocation of the subtask.

Axiom 5.24

```
∃j [  @(↓put_compress_source,j) ≤ @(↑vbi1_part1,i)
    ∧ @(↑get_compress_source,j) > @(↑vbi1_part1,i)
    ∧ @(ΩCC,j) ≤ @(↑vbi1_part1,i)]
⇒
∃k [  @(↑get_compress_source,k) > @(↑vbi1_part1,i)
    ∧ @(↓get_compress_source,k) < @(↓vbi1_part1,i)
    ∧ @(↑free_digitize,k) > @(↑vbi1_part1,i)
    ∧ @(↓free_digitize,k) < @(↓vbi1_part1,i)]
```

Axiom 5.25 represents the relationship between get_compress_source actions and free_digitize actions. The get_compress_source action occurs during the execution of the ith invocation of the VBI1 task if and only if the free_digitize action occurs. Thus, if the jth free_digitize occurs after the start of the ith VBI1 task, then the jth get_compress_source also occurs after the start of the ith VBI1 task.

Axiom 5.25

```
  @(↑free_digitize,j) > @(↑vbi1_task,i)
⇒
  @(↑get_compress_source,j) > @(↑vbi1_task,i)
```

In the second subtask of the VBI1 task, several actions are taken if a new digitize buffer can be allocated. If so, then a buffer is allocated and placed on the next_digitizing queue and a digitize operation is executed on the digitize buffer.

Axiom 5.26 represents the effect of executing the second subtask during the ith invocation of the VBI1 task. The left-hand-side of the implication is the expression described above for representing the state condition on the pool of free digitize buffers and the right-hand-side of the implication represents the fact that the set of operations performed in the conditional occur during the execution of the ith invocation of the subtask. In addition, the right-hand-side of the implication equates the frame numbers associated with each of the operations.

Axiom 5.26

```
      @(↑alloc_digitize,digitize_buffers) > @(↑vbi1_part2,i)
    ∨
      ∃m [  @(↑alloc_digitize,m+digitize_buffers) > @(↑vbi1_part2,i)
          ∧ @(↓free_digitize,m) ≤ @(↑vbi1_part2,i)]
  ⇒
    ∃k∃f [  @(↑alloc_digitize,k) > @(↑vbi1_part2,i)
          ∧ @(↓alloc_digitize,k) < @(↓vbi1_part2,i)
          ∧ @(↑digitize,k) > @(↑vbi1_part2,i)
          ∧ @(↓digitize,k) < @(↓vbi1_part2,i)
          ∧ frame(digitize,k) = f
          ∧ @(↑put_next_digitizing,k) > @(↑vbi1_part2,i)
          ∧ @(↓put_next_digitizing,k) < @(↓vbi1_part2,i)
          ∧ frame(put_next_digitizing,k) = f]
```

5.6.5 RTL Specification of the CC Task

As shown in Figure 2-24, the body of the `CC` task is a single conditional statement; several actions are taken if the `compress_sink` queue is not empty. First, a buffer is removed from the `compress_sink` queue and placed on the `transmit_video` queue. Then, a nested conditional statement is executed: if the number of buffers "in the transport system" exceeds `max_video_transport`, then a buffer is removed from the `transmit_video` queue and returned to the pool of free compress buffers (see Section 2.5.2 for the discussion of buffers "in the transport system").

I will use two axioms to describe the behavior of the `CC` task. The first describes the execution of the conditional statement, excluding the effect of executing the nested conditional. This axiom is represented in RTL as an implication using the techniques described previously. The axiom representing the execution of the nested conditional will also be an implication: the left-hand-side will be the conjunction of the conditions of the outer and inner conditional statements, while the right-hand-side will represent the execution of the body of the nested conditional.

Axiom 5.27 represents the execution of the conditional statement for the ith invocation of the `CC` task, excluding the effect of executing the nested conditional. The right-hand-side of the implication represents the fact that the set of operations performed in the conditional occur during the execution of the ith invocation of the `CC` task. In addition, the right-hand-side of the implication equates the frames numbers associated with each of the operations.

Axiom 5.27

```
    ∃j [ @(↓put_compress_sink,j) ≤ @(↑cc_task,i)
       ∧ @(↑get_compress_sink,j) > @(↑cc_task,i)]
  ⇒
   ∃k∃f [  @(↑get_compress_sink,k) > @(↑cc_task,i)
         ∧ @(↓get_compress_sink,k) < @(↓cc_task,i)
         ∧ frame(get_compress_sink,k) = f
         ∧ @(↑put_transmit,k) > @(↑cc_task,i)
         ∧ @(↓put_transmit,k) < @(↓cc_task,i)
         ∧ frame(put_transmit,k) = f]
```

The condition given in the nested conditional statement is that the number of buffers in the transport system is greater than or equal to max_video_transport at the time the nested conditional statement is executed. However, because the nested conditional statement is only executed if a compress buffer has been added to the transmit_video queue since the start of the task invocation, this test is equivalent to a test that the number of buffers in the transport system is greater than or equal to max_video_transport-1 when the task invocation begins execution. Thus, during an invocation of the CC task, the body of the nested conditional will be executed if two conditions hold when the task invocation begins execution: 1) the compress_sink queue is not empty, and 2) the number of buffers in the transport system is greater than or equal to max_video_transport-1.

In the pseudo-code descriptions of the CC and transmit_complete tasks shown in Figures 2-24 and 2-27, the variable video_transport is used to count the number of compress buffers in the transport system. This variable is incremented each time a compress buffer is added to the transmit_video queue and decremented each time a buffer is removed from the transmit_video queue and returned to the free pool. Thus, at any given time, the number of compress buffers in the transport system is equal to the difference between the number of put_transmit actions and the number of free_compress actions that have occurred up to that time. An RTL expression representing the fact that this is greater than or equal to max_video_transport-1 at the time the ith invocation of the CC task begins execution is given by the following expression:

```
  ∃j [ @(↓put_transmit,j+max_transport-1) ≤ @(↑cc_task,i)
     ∧ @(↑free_compress,j) > @(↑cc_task,i)]
```

Axiom 5.28 represents the execution of the nested conditional statement for the ith invocation of the CC task. The left-hand-side of the implication is the conjunction of the

expressions described above for representing the state condition of the `compress_sink` queue and the state condition for the number of compress buffers in the transport system. The right-hand-side of the implication represents the fact that the set of operations performed in the conditional occur during the execution of the ith invocation of the `CC` task.

Axiom 5.28

```
      ∃j [ @(↓put_compress_sink,j) ≤ @(↑cc_task,i)
         ∧ @(↑get_compress_sink,j) > @(↑cc_task,i)]
    ∧
      ∃j [ @(↓put_transmit,j+max_transport-1) ≤ @(↑cc_task,i)
         ∧ @(↑free_compress,j) > @(↑cc_task,i)]
  ⇒
    ∃k [ @(↑free_compress,k) > @(↑cc_task,i)
       ∧ @(↓free_compress,k) < @(↓cc_task,i)]
```

5.7 Bounded Delay Theorem

In the previous sections, I have presented an axiomatic specification of the application. From this specification, I now develop a proof of the correctness condition given in Figure 5-10. This proof is developed in several stages. The heart of the proof is Theorem 5.40 which will be referred to as the "main theorem" of the chapter. This theorem shows that the equation in Figure 5-13 holds for all *i*.

Recall that the `VBI1`, `VBI0`, and `CC` tasks each consisted of two conditional statements (in the `CC` task, the second was nested within the first). The equation in Figure 5-13 is divided into six groups of conjuncts corresponding to these six conditional statements. Three of these groups can be interpreted as an assertion that the body of the conditional is executed each time an invocation of the task executes (*i.e.*, the condition holds at the start of each task invocation). The first group corresponds to the second conditional statement of the `VBI1` task, the second group corresponds to the second conditional statement of the `VBI0` task, and the fourth group corresponds to the main conditional statement of the `CC` task (excluding the nested conditional statement).

The interpretation of the third group is similar to that for the first, second, and fourth. In this case however, the group of conjuncts can be interpreted as an assertion that the body of the first conditional statement in the `VBI0` task is executed during each invocation of the `VBI0` task except the first.

```
  @(↑alloc_digitize,i) > @(↑vbi1_part2,i)
∧ @(↓alloc_digitize,i) < @(↓vbi1_part2,i)
∧ @(↑digitize,i) > @(↑vbi1_part2,i)
∧ @(↓digitize,i) < @(↓vbi1_part2,i)
∧ @(↑put_next_digitizing,i) > @(↑vbi1_part2,i)
∧ @(↓put_next_digitizing,i) < @(↓vbi1_part2,i)
∧ frame(put_next_digitizing,i) = frame(digitize,i)

∧ @(↑get_next_digitizing,i) > @(↑vbi0_task,i)
∧ @(↓get_next_digitizing,i) < @(↓vbi0_task,i)
∧ @(↑put_digitizing,i) > @(↑vbi0_task,i)
∧ @(↓put_digitizing,i) < @(↓vbi0_task,i)
∧ frame(put_digitizing,i) = frame(get_next_digitizing,i)

∧ @(↑get_digitizing,i) > @(↑vbi0_task,i+1)
∧ @(↓get_digitizing,i) < @(↓vbi0_task,i+1)
∧ @(↑put_compress_source,i) > @(↑vbi0_task,i+1)
∧ @(↓put_compress_source,i) < @(↓vbi0_task,i+1)
∧ frame(put_compress_source,i) = frame(get_digitizing,i)
∧ @(↑alloc_compress,i) > @(↑vbi0_task,i+1)
∧ @(↓alloc_compress,i) < @(↓vbi0_task,i+1)
∧ @(↑put_compress_sink,i) > @(↑vbi0_task,i+1)
∧ @(↓put_compress_sink,i) < @(↓vbi0_task,i+1)
∧ frame(put_compress_sink,i) = frame(get_digitizing,i)
∧ @(↑compress,i) > @(↑vbi0_task,i+1)
∧ @(↓compress,i) < @(↓vbi0_task,i+1)
∧ frame(compress,i) = frame(get_digitizing,i)

∧ @(↑get_compress_sink,i) > @(↑cc_task,i)
∧ @(↓get_compress_sink,i) < @(↓cc_task,i)
∧ @(↑put_transmit,i) > @(↑cc_task,i)
∧ @(↓put_transmit,i) < @(↓cc_task,i)
∧ frame(put_transmit,i) = frame(get_compress_sink,i)

∧ [  i ≤ digitize_buffers
   ∨ @(↓free_digitize,i-digitize_buffers) ≤ @(↑vbi1_part2,i)]

∧ [  i ≤ max_transport
   ∨ @(↓free_compress,i-max_transport) ≤ @(↓cc_task,i)]
```

Figure 5-13: Main Theorem

The interpretation of the fifth and sixth groups of conjuncts is slightly different. The fifth group corresponds to the first subtask of the VBI1 task. Instead of asserting that each invocation of the subtask executes the body of the conditional, this group of conjuncts can be interpreted as asserting a slightly weaker property: that by the time the ith invocation of

the subtask completes execution, at least *i-digitize_buffers* `free_digitize` actions will have occurred.

The sixth group of conjuncts corresponds to the nested conditional statement in the `CC` task. However, unlike the operations in the other groups, the operations performed in the body of this conditional statement may also be performed by other tasks , (*i.e.*, removing a compress buffer from the `transmit_video` queue and returning it to the free pool may be performed by the `initiate_send` task and the `transmit_complete` task). Thus, it is not possible to assert that certain invocations of the `CC` task execute the body of the nested conditional. Instead, this group of conjuncts can be interpreted as an assertion that by the time the ith invocation of the `CC` task completes execution, at least *i-max_transport* `free_compress` actions will have occurred.

The proof of Theorem 5.40 is an induction. To support this proof, I begin by presenting several theorems for each task. One defines the time interval in which each invocation of the task is executed. The remainder specialize the general axioms about the task under certain assumptions about the events that occur prior to the execution of the task. In effect, each theorem serves as a step in the induction proof of the main theorem. The assumptions about previous events are the assumptions required from either the induction hypothesis or the previous steps of the induction proof to ensure that the task "executes correctly".

To aid the reader in following the proofs presented here, Figure 5-14 lists the page on which each axiom and theorem is defined. In addition, it gives a short intuitive description for each axiom and theorem in the chapter.

Name	Page	Intuitive Description
Axm 5.1	112	Actions start before they end
Axm 5.2	112	The ith action precedes the $i+1^{st}$ action
Thm 5.3	113	If $i \leq j$, then the ith action precedes the jth action
Axm 5.4	114	Associates frame number *i* with digitize operation *j*
Axm 5.5	114	Queues are FIFO
Axm 5.6	114	*Index* is the inverse mapping of *frame*
Axm 5.7	114	Period of the VBI logical interrupt
Axm 5.8	114	Relationship between `start_compress` and CC interrupts
Axm 5.9	115	Messages from DVI handler to `VBI` task
Axm 5.10	115	Messages from DVI handler to `CC` task
Axm 5.11	116	Tasks execute prior to their deadline
Axm 5.12	116	Subtasks of `VBI1` task execute in order
Axm 5.13	116	Mutual exclusion
Axm 5.14	117	The ith AMO action starts after the ith task invocation
Axm 5.15	117	AMO actions do not occur between task invocations
Axm 5.16	118	At most one AMO action per task invocation
Thm 5.17	118	Which task invocation executes first AMO action?
Thm 5.18	119	If i-1st AMO action occurs in one invocation, i^{th} occurs in next
Axm 5.19	122	Odd-numbered invocations of the `VBI` task send to `VBI1` task
Axm 5.20	122	Even-numbered invocations of the `VBI` task send to `VBI0` task
Axm 5.21	123	First conditional statement in the `VBI0` task (if executed)
Axm 5.22	123	First conditional statement in the `VBI0` task (if not executed)
Axm 5.23	124	Second conditional statement in the `VBI0` task
Axm 5.24	125	First conditional statement in the `VBI1` task
Axm 5.25	125	`free_digitize` implies `get_compress_source`
Axm 5.26	126	Second conditional statement in the `VBI1` task
Axm 5.27	127	Main conditional statement in the `CC` task
Axm 5.28	128	Nested conditional statement in the `CC` task
Thm 5.29	132	The `VBI` task executes in a specific interval
Thm 5.30	132	The `VBI0` task executes in a specific interval
Thm 5.31	133	Second conditional statement in the `VBI0` task (induction step)
Lem 5.32	135	First `get_digitizing` executed in second `VBI0` task
Thm 5.33	137	First conditional statement in the `VBI0` task (induction step)
Thm 5.34	140	The `VBI1` task executes in a specific interval
Thm 5.35	141	First conditional statement in the `VBI1` task (induction step)
Thm 5.36	143	Second conditional statement in the `VBI1` task (induction step)
Thm 5.37	145	The `CC` task executes in a specific interval
Thm 5.38	146	Main conditional statement in the `CC` task (induction step)
Thm 5.39	147	Nested conditional statement in the `CC` task (induction step)
Thm 5.40	149	Main theorem
Thm 5.43	160	Bounded delay and correctness condition

Figure 5-14: Summary of Axioms and Theorems

5.7.1 Theorems for the VBI Task

First, I develop a theorem that uses the axioms that represent the behavior of the VBI logical interrupt along with those that represent the scheduling and execution of tasks to

define two properties of the VBI task: the interval within which each invocation of the task executes, and an upper bound on its logical arrival time.

Theorem 5.29

$$\begin{aligned} &@(\uparrow \texttt{vbi_task},i) > (i-1)\cdot \mathit{vbi_period} \\ \wedge\ &@(\downarrow \texttt{vbi_task},i) < i\cdot \mathit{vbi_period} \\ \wedge\ &@(\downarrow \texttt{logical_send_vbi},i) \le (i-1)\cdot \mathit{vbi_period} + \mathit{dvi_delay} \end{aligned}$$

Proof: First, by Axioms 5.11, 5.1, 5.9, and 5.7

$$\begin{aligned} @(\uparrow \texttt{vbi_task},i) &\ge @(\downarrow \texttt{send_vbi},i) \\ &> @(\uparrow \texttt{send_vbi},i) \\ &> @(\Omega \texttt{VBI},i) \\ &> (i-1)\cdot \mathit{vbi_period} \end{aligned} \tag{5.5}$$

Next, by Axioms 5.11, 5.9, 5.7, and the bound on *vbi_period* given in Figure 5-2

$$\begin{aligned} @(\downarrow \texttt{vbi_task},i) &< @(\downarrow \texttt{logical_send_vbi},i) + \mathit{vbi_deadline} \\ &< @(\Omega \texttt{VBI},i) + \mathit{dvi_delay} + \mathit{vbi_deadline} \\ &< (i-1)\cdot \mathit{vbi_period} + \mathit{dvi_delay} + \mathit{vbi_deadline} \\ &< (i-1)\cdot \mathit{vbi_period} + \mathit{vbi_period} \\ &< i\cdot \mathit{vbi_period} \end{aligned} \tag{5.6}$$

Finally, by Axioms 5.9 and 5.7

$$\begin{aligned} @(\downarrow \texttt{logical_send_vbi},i) &\le @(\Omega \texttt{VBI},i) + \mathit{dvi_delay} \\ &\le (i-1)\cdot \mathit{vbi_period} + \mathit{dvi_delay} \end{aligned} \tag{5.7}$$

Together, (5.5), (5.6), and (5.7) prove the theorem. □

5.7.2 Theorems for the VBI0 Task

The next theorem defines the interval within which each invocation of the VBI0 task executes.

Theorem 5.30

$$\begin{aligned} &@(\uparrow \texttt{vbi0_task},i) > (2i-1)\cdot \mathit{vbi_period} \\ \wedge\ &@(\downarrow \texttt{vbi0_task},i) < 2i\cdot \mathit{vbi_period} \end{aligned}$$

Proof: By Axioms 5.11, 5.1, and 5.20, and Theorem 5.29

$$\begin{aligned} @(\uparrow \texttt{vbi0_task},i) &\ge @(\downarrow \texttt{send_vbi0},i) \\ &> @(\uparrow \texttt{send_vbi0},i) \\ &> @(\uparrow \texttt{vbi_task},2i) \\ &> (2i-1)\cdot \mathit{vbi_period} \end{aligned} \tag{5.8}$$

By Axioms 5.11, 5.20, 5.9, and 5.7, and the bound on *vbi_period* given in Figure 5-2

```
@(↓vbi0_task,i) < @(↓logical_send_vbi0,i) + vbi_deadline
                < @(↓logical_send_vbi,2i) + vbi_deadline
                < @(ΩVBI,2i) + dvi_delay + vbi_deadline
                < (2i-1)·vbi_period + dvi_delay + vbi_deadline
                < 2i·vbi_period                                  (5.9)
```

Together (5.8) and (5.9) prove the theorem. □

The next two theorems address the effect of executing an invocation of the `VBI0` task under several assumptions about the events that preceded the start of the invocation. Each theorem addresses the effect of one of the conditional statements.

The second conditional statement in the `VBI0` task is represented by Axiom 5.23. Theorem 5.31 specializes this axiom for an assumption that two events have occurred prior to the start of the ith invocation of the task:

1. the ith `put_next_digitizing` preceded the start of the ith invocation of the `VBI0` task.
2. unless this is the first invocation of the `VBI0` task, the i-1st `get_next_digitizing` was performed by the i-1st invocation of the `VBI0` task.

The theorem shows that if these events occur prior to the start of the invocation, then the ith instance of each of the operations in the body of the conditional is executed during the invocation.

Theorem 5.31

```
    @(↓put_next_digitizing,i) ≤ @(↑vbi0_task,i)
  ∧ [  i = 1
     ∨ [  @(↑get_next_digitizing,i-1) > @(↑vbi0_task,i-1)
        ∧ @(↓get_next_digitizing,i-1) < @(↓vbi0_task,i-1)]]
⇒
    @(↑get_next_digitizing,i) > @(↑vbi0_task,i)
  ∧ @(↓get_next_digitizing,i) < @(↓vbi0_task,i)
  ∧ @(↑put_digitizing,i) > @(↑vbi0_task,i)
  ∧ @(↓put_digitizing,i) < @(↓vbi0_task,i)
  ∧ frame(put_digitizing,i) = frame(get_next_digitizing,i)
```

Proof: Assume the l.h.s. of the implication. By this assumption and Axiom 5.14

```
  @(↓put_next_digitizing,i) ≤ @(↑vbi0_task,i)
∧ @(↑get_next_digitizing,i) > @(↑vbi0_task,i)
```

Thus, by Axiom 5.23 there exist k and f such that equation (5.10) holds. Choose k and f.

```
  @(↑get_next_digitizing,k) > @(↑vbi0_task,i)
∧ @(↓get_next_digitizing,k) < @(↓vbi0_task,i)
∧ frame(get_next_digitizing,k) = f
∧ @(↑put_digitizing,k) > @(↑vbi0_task,i)
∧ @(↓put_digitizing,k) < @(↓vbi0_task,i)
∧ frame(put_digitizing,k) = f                                    (5.10)
```

I now show that $k = i$. There are two cases, depending on i.

Case 1: Assume $i = 1$. By Axiom 5.14 and equation (5.10)

```
  @(↑get_next_digitizing,1) > @(↑vbi0_task,1)
∧ @(↑get_next_digitizing,k) > @(↑vbi0_task,1)
∧ @(↓get_next_digitizing,k) < @(↓vbi0_task,1)
```

and thus by Theorem 5.17, $k = i$.

Case 2: Assume $i > 1$. By the l.h.s. of the theorem and equation (5.10)

```
  @(↑get_next_digitizing,i-1) > @(↑vbi0_task,i-1)
∧ @(↓get_next_digitizing,i-1) < @(↓vbi0_task,i-1)
∧ @(↑get_next_digitizing,k) > @(↑vbi0_task,i)
∧ @(↓get_next_digitizing,k) < @(↓vbi0_task,i)
```

and thus by Theorem 5.18, $k = i$.

Thus, in either case, $k = i$. Substituting i for k in equation (5.10)

```
  @(↑get_next_digitizing,i) > @(↑vbi0_task,i)
∧ @(↓get_next_digitizing,i) < @(↓vbi0_task,i)
∧ @(↑put_digitizing,i) > @(↑vbi0_task,i)
∧ @(↓put_digitizing,i) < @(↓vbi0_task,i)                         (5.11)
```

Also, in equation (5.10)

```
frame(put_digitizing,k) = f = frame(get_next_digitizing,k)   (5.12)
```

Together, (5.11) and (5.12) form the r.h.s. of the implication, proving the theorem. □

The first conditional statement in the `VBI0` task is represented by Axiom 5.21. Theorem 5.33 specializes this axiom for an assumption that several events have occurred prior to the start of the i+1st invocation of the task:

1. the ith `put_digitizing` action was performed by the ith invocation of the `VBI0` task.

2. unless this is the second invocation of the `VBI0` task, the i-1st `get_digitizing` action was performed by the ith invocation of the `VBI0` task.

3. if the `VBI0` task has already been executed at least *compress_buffers* times, then at least *i-compress_buffers* compress buffers have already been returned to the free pool.

4. unless this is the second invocation of the `VBI0` task, the i-1st `alloc_compress` action was performed by the ith invocation of the `VBI0` task.

The theorem shows that if these events occur prior to the start of the invocation, then the ith instance of each of the operations in the body of the conditional is executed during the i+1st invocation of the `VBI0` task.

Before proving Theorem 5.33, I prove a lemma. This lemma shows that the first instance of the `get_digitizing` action and the first instance of the `alloc_compress` action do not occur until at least the second invocation of the `VBI0` task.

Lemma 5.32

```
   @(↑get_digitizing,1) > @(↑vbi0_task,2)
 ∧ @(↑alloc_compress,1) > @(↑vbi0_task,2)
```

Proof: By Theorem 5.3, and Axioms 5.1 and 5.14

```
@(↓put_digitizing,j) ≥ @(↓put_digitizing,1)
                     > @(↑put_digitizing,1)
                     > @(↑vbi0_task,1)
```

This can be combined with an arbitrary clause in a disjunction and generalized to form

```
∀j [  @(↓put_digitizing,j) > @(↑vbi0_task,1)
    ∨ @(↑get_digitizing,j) ≤ @(↑vbi0_task,1)]
```

which is equivalent to

```
~∃j [  @(↓put_digitizing,j) ≤ @(↑vbi0_task,1)
     ∧ @(↑get_digitizing,j) > @(↑vbi0_task,1)]
```

By Axiom 5.22

```
∀k [  @(↓get_digitizing,k) < @(↑vbi0_task,1)
    ∨ @(↑get_digitizing,k) > @(↓vbi0_task,1)]              (5.13)
```

and

```
∀k [  @(↓alloc_compress,k) < @(↑vbi0_task,1)
    ∨ @(↑alloc_compress,k) > @(↓vbi0_task,1)]              (5.14)
```

By Axioms 5.1 and 5.14

```
@(↓get_digitizing,1) > @(↑get_digitizing,1)
                     > @(↑vbi0_task,1)
```

and thus for $k = 1$ the first disjunct in (5.13) does not hold. Thus, the second disjunct holds.

```
@(↑get_digitizing,1) > @(↓vbi0_task,1)
```

and by Axiom 5.15

```
@(↑get_digitizing,1) > @(↑vbi0_task,2)                     (5.15)
```

Similarly by Axioms 5.1 and 5.14, equation (5.14) and Axiom 5.15

```
@(↑alloc_compress,1) > @(↑vbi0_task,2)                     (5.16)
```

Together, (5.15) and (5.16) show the lemma. □

Theorem 5.33

```
      @(↑put_digitizing,i) > @(↑vbi0_task,i)
    ∧ @(↓put_digitizing,i) < @(↓vbi0_task,i)
    ∧ [  i = 1
       ∨ [  @(↑get_digitizing,i-1) > @(↑vbi0_task,i)
          ∧ @(↓get_digitizing,i-1) < @(↓vbi0_task,i)]]
    ∧ [  i ≤ compress_buffers
       ∨ @(↓free_compress,i-compress_buffers) ≤ @(↑vbi0_task,i+1)]
    ∧ [  i = 1
       ∨ [  @(↑alloc_compress,i-1) > @(↑vbi0_task,i)
          ∧ @(↓alloc_compress,i-1) < @(↓vbi0_task,i)]]
  ⇒
      @(↑get_digitizing,i) > @(↑vbi0_task,i+1)
    ∧ @(↓get_digitizing,i) < @(↓vbi0_task,i+1)
    ∧ @(↑put_compress_source,i) > @(↑vbi0_task,i+1)
    ∧ @(↓put_compress_source,i) < @(↓vbi0_task,i+1)
    ∧ frame(put_compress_source,i) = frame(get_digitizing,i)
    ∧ @(↑alloc_compress,i) > @(↑vbi0_task,i+1)
    ∧ @(↓alloc_compress,i) < @(↓vbi0_task,i+1)
    ∧ @(↑put_compress_sink,i) > @(↑vbi0_task,i+1)
    ∧ @(↓put_compress_sink,i) < @(↓vbi0_task,i+1)
    ∧ frame(put_compress_sink,i) = frame(get_digitizing,i)
    ∧ @(↑compress,i) > @(↑vbi0_task,i+1)
    ∧ @(↓compress,i) < @(↓vbi0_task,i+1)
    ∧ frame(compress,i) = frame(get_digitizing,i)
```

Proof: Assume the l.h.s. of the implication. The proof consists of three steps. In steps 1 and 2, I show that equations (5.17) and (5.18) can be derived from the l.h.s. of the theorem. In step 3, I use Axiom 5.21 to show that the right-hand-side of the theorem holds.

Step 1: I begin by showing that equation (5.17) holds.

```
    ∃j [  @(↓put_digitizing,j) ≤ @(↑vbi0_task,i+1)
        ∧ @(↑get_digitizing,j) > @(↑vbi0_task,i+1)]          (5.17)
```

By the l.h.s. of the theorem and Axiom 5.2

```
  @(↓put_digitizing,i) < @(↓vbi0_task,i)
                       < @(↑vbi0_task,i+1)
                       ≤ @(↑vbi0_task,i+1)
```

Next, there are two cases, depending on *i*.

Case 1: Assume $i = 1$. By Lemma 5.32

```
@(↑get_digitizing,1) > @(↑vbi0_task,2)
```

Case 2: Assume $i > 1$. By the l.h.s. of the theorem

```
@(↑get_digitizing,i-1) > @(↑vbi0_task,i)
```

and thus by Axiom 5.16

```
@(↑get_digitizing,i) > @(↑vbi0_task,i+1)
```

Thus in either case equation (5.17) holds.

Step 2: Next, I show that equation (5.18) holds.

```
  @(↑alloc_compress,compress_buffers) > @(↑vbi0_task,i+1)
∨
  ∃m [  @(↑alloc_compress,m+compress_buffers) > @(↑vbi0_task,i+1)
      ∧ @(↓free_compress,m) ≤ @(↑vbi0_task,i+1)]           (5.18)
```

There are three cases, depending on *i*.

Case 1: Assume $i = 1$. By Lemma 5.32

```
@(↑alloc_compress,1) > @(↑vbi0_task,2)
```

Then by Axiom 5.16, the bound on *compress_buffers* given in Figure 5-2, and Theorem 5.3

```
@(↑alloc_compress,compress_buffers) > @(↑vbi0_task,compress_buffers+1)
                                    > @(↑vbi0_task,2)
```

Case 2: Assume $1 < i \le$ *compress_buffers*. By the l.h.s. of the theorem

```
@(↑alloc_compress,i-1) > @(↑vbi0_task,i)
```

Thus, by Axiom 5.16

```
@(↑alloc_compress,compress_buffers) > @(↑vbi0_task,compress_buffers+1)
                                    > @(↑vbi0_task,i+1)
```

Case 3: Assume $i >$ *compress_buffers*. By the l.h.s. of the theorem

```
  @(↑alloc_compress,i-1) > @(↑vbi0_task,i)
∧ @(↓free_compress,i-compress_buffers) ≤ @(↑vbi0_task,i+1)
```

Let $m = i$ - *compress_buffers*. By Axiom 5.16

```
  @(↑alloc_compress,m+compress_buffers) > @(↑vbi0_task,i+1)
∧ @(↓free_compress,m) ≤ @(↑vbi0_task,i+1)
```

Thus in each case equation (5.18) holds.

Step 3: Together, equations (5.17) and (5.18) are the l.h.s. of Axiom 5.21. Thus, there exist k and f such that equation (5.19) holds. Choose k and f.

```
  @(↑get_digitizing,k) > @(↑vbi0_task,i+1)
∧ @(↓get_digitizing,k) < @(↓vbi0_task,i+1)
∧ frame(get_digitizing,k) = f
∧ @(↑put_compress_source,k) > @(↑vbi0_task,i+1)
∧ @(↓put_compress_source,k) < @(↓vbi0_task,i+1)
∧ frame(put_compress_source,k) = f
∧ @(↑alloc_compress,k) > @(↑vbi0_task,i+1)
∧ @(↓alloc_compress,k) < @(↓vbi0_task,i+1)
∧ @(↑put_compress_sink,k) > @(↑vbi0_task,i+1)
∧ @(↓put_compress_sink,k) < @(↓vbi0_task,i+1)
∧ frame(put_compress_sink,k) = f
∧ @(↑compress,k) > @(↑vbi0_task,i+1)
∧ @(↓compress,k) < @(↓vbi0_task,i+1)
∧ frame(compress,k) = f                                     (5.19)
```

There are two cases, depending on i.

Case 1: Assume $i = 1$. By Lemma 5.32 and equation (5.19)

```
  @(↑get_digitizing,1) > @(↑vbi0_task,i+1)
∧ @(↑get_digitizing,k) > @(↑vbi0_task,i+1)
∧ @(↓get_digitizing,k) < @(↓vbi0_task,i+1)
```

and thus by Theorem 5.17, $k = i$.

Case 2: Assume $i > 1$. By the l.h.s. of the theorem and equation (5.19)

```
  @(↑get_digitizing,i-1) > @(↑vbi0_task,i)
∧ @(↓get_digitizing,i-1) < @(↓vbi0_task,i)
∧ @(↑get_digitizing,k) > @(↑vbi0_task,i+1)
∧ @(↓get_digitizing,k) < @(↓vbi0_task,i+1)
```

and thus by Theorem 5.18, $k = i$. Thus in either case $k = i$.

Furthermore, in equation (5.19)

```
frame(get_digitizing,k) = frame(put_compress_sink,k)
                        = frame(compress,k)
                        = frame(put_compress_source,k)
                        = f
```

Substituting i for k and `frame(get_digitizing,k)` for f in equation (5.19) yields the right-hand-side of the theorem. This proves the theorem. □

5.7.3 Theorems for the VBI1 Task

The next theorem defines the interval within which each invocation of the `VBI1` task executes.

Theorem 5.34

```
  @(↑vbi1_task,i) > (2i-2)·vbi_period
∧ @(↓vbi1_task,i) < (2i-1)·vbi_period
```

Proof: By Axioms 5.11, 5.1 and 5.19, and Theorem 5.29

```
@(↑vbi1_task,i) ≥ @(↓send_vbi1,i)
                > @(↑send_vbi1,i)
                > @(↑vbi_task,2i-1)
                > (2i-2)·vbi_period                    (5.20)
```

By Axioms 5.11, 5.19, 5.9, and 5.7, and the bound on *vbi_period* given in Figure 5-2

```
@(↓vbi1_task,i) < @(↓logical_send_vbi1,i) + vbi_deadline
                < @(↓logical_send_vbi,2i-1) + vbi_deadline
                < @(ΩVBI,2i-1) + dvi_delay + vbi_deadline
                < (2i-2)·vbi_period + dvi_delay + vbi_deadline
                < (2i-1)·vbi_period                    (5.21)
```

Together (5.20) and (5.21) prove the theorem. □

The next two theorems address the effect of executing an invocation of the `VBI1` task under several assumptions about the events that preceded the start of the invocation. Each theorem address the effect of executing one of the two subtasks of the `VBI1` task.

The effect of executing the first subtask of the `VBI1` task is represented by Axiom 5.24. Theorem 5.35 specializes this axiom for an assumption that several events have occurred prior to the start of the kth invocation of the subtask, where $k = i + digitize_buffers$:

1. the ith `put_compress_source` preceded the start of the first subtask of the kth invocation of the `VBI1` task.

2. the ith CC interrupt preceded the start of the first subtask of the kth invocation of the `VBI1` task.

3. if the `VBI1` task has already executed at least *digitize_buffers*-1 times prior to the start of the first subtask of the kth invocation of the `VBI1` task, then at least i-1 digitize buffers had been returned to the free pool prior to the end of the first subtask of the k-1st invocation of the `VBI1` task.

The theorem shows that if these events occur prior to the start of the kth subtask, then the ith `free_digitize` action occurs prior to the end of the kth subtask.

Theorem 5.35

```
     @(↓put_compress_source,i) ≤ @(↑vbi1_part1,i+digitize_buffers)
   ∧ @(ΩCC,i) ≤ @(↑vbi1_part1,i+digitize_buffers)
   ∧ [  i = 1
      ∨ @(↓free_digitize,i-1) < @(↓vbi1_part1,i+digitize_buffers-1)]
⇒
   @(↓free_digitize,i) < @(↓vbi1_part1,i+digitize_buffers)
```

Proof: Assume the l.h.s. of the implication. There are two cases depending on whether or not the ith `free_digitize` action started prior to the start of the task.

Case 1: Assume

```
@(↑free_digitize,i) ≤ @(↑vbi1_task,i+digitize_buffers)
```

By Axiom 5.12

```
@(↑free_digitize,i) ≤ @(↑vbi1_task,i+digitize_buffers)
                    < @(↑vbi1_part1,i+digitize_buffers)
```

and thus by the contrapositive of Axiom 5.15 and Theorem 5.3

```
@(↓free_digitize,i) < @(↓vbi1_part1,i+digitize_buffers-1)
                    < @(↓vbi1_part1,i+digitize_buffers)
```

Case 2: Assume

```
@(↑free_digitize,i) > @(↑vbi1_task,i+digitize_buffers)
```

Thus by Axioms 5.25 and 5.12

```
@(↑get_compress_source,i) > @(↑vbi1_task,i+digitize_buffers)
                          > @(↑vbi1_part1,i+digitize_buffers)
```

Combining this with the l.h.s. of the theorem yields

```
  @(↓put_compress_source,i) ≤ @(↑vbi1_part1,i+digitize_buffers)
∧ @(↑get_compress_source,i) > @(↑vbi1_part1,i+digitize_buffers)
∧ @(ΩCC,i) ≤ @(↑vbi1_part1,i+digitize_buffers)
```

which is the l.h.s. of Axiom 5.24. Thus, by Axiom 5.24, there exist k and f such that equation (5.22) holds. Choose k and f.

```
  @(↑free_digitize,k) > @(↑vbi1_part1,i+digitize_buffers)
∧ @(↓free_digitize,k) < @(↓vbi1_part1,i+digitize_buffers)     (5.22)
```

I now show that $k \geq i$. There are two cases, depending on i.

Case 2a: Assume $i = 1$. Then $k \geq i$.

Case 2b: Assume $i > 1$.

By equation (5.22), Axiom 5.2, the l.h.s. of the theorem, and Axiom 5.1

```
@(↑free_digitize,k) > @(↑vbi1_part1,i+digitize_buffers)
                    > @(↓vbi1_part1,i+digitize_buffers-1)
                    > @(↓free_digitize,i-1)
                    > @(↑free_digitize,i-1)
```

Thus, by the contrapositive of Theorem 5.3, $k > i - 1$.

Thus in both case 2a and 2b, $k \geq i$. By Theorem 5.3 and equation (5.22)

```
@(↓free_digitize,i) ≤ @(↓free_digitize,k)
                    ≤ @(↓vbi1_part1,i+digitize_buffers)
```

This proves the theorem. □

The effect of executing the second subtask of the VBI1 task is represented by Axiom 5.26. Theorem 5.36 specializes this axiom for an assumption that several events have occurred prior to the start of the ith invocation of the subtask:

1. if the second subtask of the ith invocation of the VBI1 task has already executed at least *digitize_buffers* times, then at least *i-digitize_buffers* digitize buffers have already been returned to the free pool.

2. unless this is the second subtask of the first invocation of the VBI1 task, the i-1st `alloc_digitize` was performed by the second subtask of the i-1st invocation of the VBI1 task.

The theorem shows that if these events occur prior to the start of the subtask, then the ith instance of each of the operations in the body of the conditional is executed during the ith subtask.

Theorem 5.36

```
     [  i ≤ digitize_buffers
      ∨ @(↓free_digitize,i-digitize_buffers) ≤ @(↑vbi1_part2,i)]
   ∧ [  i = 1
      ∨ [  @(↑alloc_digitize,i-1) > @(↑vbi1_part2,i-1)
         ∧ @(↓alloc_digitize,i-1) < @(↓vbi1_part2,i-1)]]
 ⇒
     @(↑alloc_digitize,i) > @(↑vbi1_part2,i)
   ∧ @(↓alloc_digitize,i) < @(↓vbi1_part2,i)
   ∧ @(↑digitize,i) > @(↑vbi1_part2,i)
   ∧ @(↓digitize,i) < @(↓vbi1_part2,i)
   ∧ @(↑put_next_digitizing,i) > @(↑vbi1_part2,i)
   ∧ @(↓put_next_digitizing,i) < @(↓vbi1_part2,i)
   ∧ frame(put_next_digitizing,i) = frame(digitize,i)
```

Proof: Assume the l.h.s. of the implication. The proof consists of two steps. In the first step, I show that equation (5.23) can be derived from the l.h.s. of the theorem. In the second step, I use Axiom 5.26 to show that the right-hand-side of the theorem holds.

Step 1: I begin by showing that equation (5.23) holds.

```
   @(↑alloc_digitize,digitize_buffers) > @(↑vbi1_part2,i)
 ∨
   ∃m [  @(↑alloc_digitize,m+digitize_buffers) > @(↑vbi1_part2,i)
       ∧ @(↓free_digitize,m) ≤ @(↑vbi1_part2,i)]           (5.23)
```

There are two cases depending on i.

Case 1: Assume $i \leq$ *digitize_buffers*. By Axiom 5.14

```
 (↑alloc_digitize,i) > @(↑vbi1_part2,i)
```

and thus by Axiom 5.16 and Theorem 5.3

```
 @(↑alloc_digitize,digitize_buffers) > @(↑vbi1_part2,digitize_buffers)
                                     > @(↑vbi1_part2,i)
```

Case 2: Assume $i >$ *digitize_buffers*. Let $m = i$ - *digitize_buffers*. By Axiom 5.14 and the l.h.s. of the theorem

```
  @(↑alloc_digitize,i) > @(↑vbi1_part2,i)
∧ @(↓free_compress,i-digitize_buffers) ≤ @(↑vbi1_part2,i)
```

which expressed in terms of *m* is

```
  @(↑alloc_digitize,m+digitize_buffers) > @(↑vbi1_part2,i)
∧ @(↓free_compress,m) ≤ @(↑vbi1_part2,i)
```

Thus in each case (5.23) holds.

Step 2: Equation (5.23) is the l.h.s. of Axiom 5.26. Thus, there exist *k* and *f* such that equation (5.24) holds. Choose *k* and *f*.

```
  @(↑alloc_digitize,k) > @(↑vbi1_part2,i)
∧ @(↓alloc_digitize,k) < @(↓vbi1_part2,i)
∧ @(↑digitize,k) > @(↑vbi1_part2,i)
∧ @(↓digitize,k) < @(↓vbi1_part2,i)
∧ frame(digitize,k) = f
∧ @(↑put_next_digitizing,k) > @(↑vbi1_part2,i)
∧ @(↓put_next_digitizing,k) < @(↓vbi1_part2,i)
∧ frame(put_next_digitizing,k) = f                          (5.24)
```

There are two cases, depending on *i*.

Case 1: Assume $i = 1$. By Axiom 5.14 and equation (5.24)

```
  @(↑alloc_digitize,1) > @(↑vbi1_part2,1)
∧ @(↑alloc_digitize,k) > @(↑vbi1_part2,1)
∧ @(↓alloc_digitize,k) < @(↓vbi1_part2,1)
```

and thus by Theorem 5.17, $k = i$.

Case 2: Assume $i > 1$. By the l.h.s. of the theorem and equation (5.24)

```
  @(↑alloc_digitize,i-1) > @(↑vbi1_part2,i-1)
∧ @(↓alloc_digitize,i-1) < @(↓vbi1_part2,i-1)
∧ @(↑alloc_digitize,k) > @(↑vbi1_part2,i)
∧ @(↓alloc_digitize,k) < @(↓vbi1_part2,i)
```

and thus by Theorem 5.18, $k = i$. Thus in either case $k = i$.

Furthermore, in equation (5.24)

```
frame(put_next_digitizing,k) = frame(digitize,k) = f
```

Substituting i for k and `frame(digitize,k)` for f in equation (5.24) yields the right-hand-side of the theorem. This proves the theorem. ☐

5.7.4 Theorems for the CC Task

The next theorem defines the interval within which each invocation of the CC task executes.

Theorem 5.37

$$@(\uparrow \texttt{cc_task,i}) > @(\downarrow \texttt{compress,i}) \wedge @(\downarrow \texttt{cc_task,i}) < @(\downarrow \texttt{compress,i}) + 3 \cdot vbi_period$$

Proof: By Axioms 5.11, 5.1, 5.10, and 5.8

$$\begin{aligned} @(\uparrow \texttt{cc_task,i}) &\geq @(\downarrow \texttt{send_cc,i}) \\ &> @(\uparrow \texttt{send_cc,i}) \\ &> @(\Omega \texttt{CC,i}) \\ &> @(\downarrow \texttt{compress,i}) \end{aligned} \quad (5.25)$$

By Axioms 5.11, 5.10 and 5.8, and the bound on *compress_request* given in Figure 5-2

$$\begin{aligned} @(\downarrow \texttt{cc_task,i}) &< @(\downarrow \texttt{logical_send_cc,i}) + cc_deadline \\ &< @(\Omega \texttt{CC,i}) + dvi_delay + cc_deadline \\ &< @(\downarrow \texttt{compress,i}) + compress_request + vbi_period \\ &< @(\downarrow \texttt{compress,i}) + 3 \cdot vbi_period \end{aligned} \quad (5.26)$$

Together (5.25) and (5.26) prove the theorem. ☐

The next two theorems address the effect of executing an invocation of the CC task under several assumptions about the events that preceded the start of the invocation. Each theorem addresses the effect of one of the conditional statements.

The main conditional statement in the CC task (excluding the nested conditional) is represented by Axiom 5.27. Theorem 5.38 specializes this axiom for an assumption that several events have occurred prior to the start of the ith invocation of the task:

1. the ith `put_compress_sink` preceded the start of the ith invocation of the CC task.

2. unless this is the first invocation of the CC task, the i-1st `get_compress_sink` was performed by the i-1st invocation of the VBI0 task.

The theorem shows that if these events occur prior to the start of the invocation, then the ith instance of each of the operations in the body of the conditional is executed during the invocation.

Theorem 5.38

```
      @(↓put_compress_sink,i) ≤ @(↑cc_task,i)
    ∧ [   i = 1
        ∨ [  @(↑get_compress_sink,i-1) > @(↑cc_task,i-1)
           ∧ @(↓get_compress_sink,i-1) < @(↓cc_task,i-1)]]
  ⇒
      @(↑get_compress_sink,i) > @(↑cc_task,i)
    ∧ @(↓get_compress_sink,i) < @(↓cc_task,i)
    ∧ @(↑put_transmit,i) > @(↑cc_task,i)
    ∧ @(↓put_transmit,i) < @(↓cc_task,i)
    ∧ frame(put_transmit,i) = frame(get_compress_sink,i)
```

Proof: Assume the l.h.s. of the implication. By this assumption and Axiom 5.14

```
    @(↓put_compress_sink,i) ≤ @(↑cc_task,i)
  ∧ @(↑get_compress_sink,i) > @(↑cc_task,i)
```

Thus, by Axiom 5.27 there exist k and f such that equation (5.27) holds. Choose k and f.

```
    @(↑get_compress_sink,k) > @(↑cc_task,i)
  ∧ @(↓get_compress_sink,k) < @(↓cc_task,i)
  ∧ frame(get_compress_sink,k) = f
  ∧ @(↑put_transmit,k) > @(↑cc_task,i)
  ∧ @(↓put_transmit,k) < @(↓cc_task,i)
  ∧ frame(put_transmit,k) = f                                   (5.27)
```

I now show that $k = i$. There are two cases, depending on i.

Case 1: Assume $i = 1$. By Axiom 5.14 and equation (5.27)

```
    @(↑get_compress_sink,1) > @(↑cc_task,i)
  ∧ @(↑get_compress_sink,k) > @(↑cc_task,i)
  ∧ @(↓get_compress_sink,k) < @(↓cc_task,i)
```

and by Theorem 5.17, $k = i$.

Case 2: Assume $i > 1$. By the l.h.s. of the theorem and equation (5.27)

```
    @(↑get_compress_sink,i-1) > @(↑cc_task,i-1)
  ∧ @(↓get_compress_sink,i-1) < @(↓cc_task,i-1)
  ∧ @(↑get_compress_sink,k) > @(↑cc_task,i)
  ∧ @(↓get_compress_sink,k) < @(↓cc_task,i)
```

and thus by Theorem 5.18, $k = i$. Thus in either case $k = i$.

Furthermore, in equation (5.27)

```
frame(put_transmit,k) = frame(get_compress_sink,k) = f
```

Substituting i for k and `frame(get_compress_sink,k)` for f in equation (5.27) yields the right-hand-side of the theorem. This proves the theorem. □

The nested conditional statement in the `CC` task is represented by Axiom 5.28. Theorem 5.39 specializes this axiom for an assumption that several events have occurred prior to the start of the kth invocation of the task, where $k = i + max_transport$:

1. the kth `put_compress_sink` preceded the start of the kth invocation of the `CC` task.

2. the kth `get_compress_sink` occurred after the start of the kth invocation of the `CC` task.

3. the k-1st `put_transmit` preceded the start of the kth invocation of the `CC` task.

4. if the `CC` task has already executed at least *max_transport* times prior to the start of the kth invocation of the `CC` task, then at least i-1 compress buffers had been returned to the free pool prior to the end of the k-1st invocation of the `CC` task.

The theorem shows that if these events occur prior to the start of the kth subtask, then the ith `free_compress` action occurs prior to the end of the kth subtask.

Theorem 5.39

```
    @(↓put_compress_sink,i+max_transport) ≤ @(↑cc_task,i+max_transport)
  ∧ @(↑get_compress_sink,i+max_transport) > @(↑cc_task,i+max_transport)
  ∧ @(↓put_transmit,i+max_transport-1) ≤ @(↑cc_task,i+max_transport)
  ∧ [  i = 1
     ∨ @(↓free_compress,i-1) ≤ @(↓cc_task,i+max_transport-1)]
⇒
   @(↓free_compress,i) ≤ @(↓cc_task,i+max_transport)
```

Proof: Assume the l.h.s. of the implication. There are two cases depending on whether or not the ith `free_compress` action started prior to the start of the task.

Case 1: Assume

```
@(↑free_compress,i) ≤ @(↑cc_task,i+max_transport)
```

There are two subcases depending on whether the ith `free_compress` action was executed by an invocation of the `cc_task` or the `tc_task`.

Case 1a: Assume there exists a *j* such that

```
  @(↑free_compress,i) > @(↑cc_task,j)
∧ @(↓free_compress,i) < @(↓cc_task,j)
```

In this case, $j < i+max_transport$. Thus by Theorem 5.3

```
@(↓free_compress,i) < @(↓cc_task,j)
                    < @(↓cc_task,i+max_transport)
```

Case 1b: Assume there exists a *j* such that

```
  @(↑free_compress,i) > @(↑tc_task,j)
∧ @(↓free_compress,i) < @(↓tc_task,j)
```

In this case,

```
@(↑tc_task,j) < @(↑free_compress,i)
              < @(↑cc_task,i+max_transport)
```

and thus by Axioms 5.13

```
@(↓tc_task,j) < @(↑cc_task,i+max_transport)
```

Combining these expressions and applying Axiom 5.1 yields

```
@(↓free_compress,i) < @(↓tc_task,j)
                    < @(↑cc_task,i+max_transport)
                    < @(↓cc_task,i+max_transport)
```

Case 2: Assume

```
@(↑free_compress,i) > @(↑cc_task,i+max_transport)
```

Combining this with the l.h.s. of the theorem yields

```
  @(↓put_compress_sink,i+max_transport) ≤ @(↑cc_task,i+max_transport)
∧ @(↑get_compress_sink,i+max_transport) > @(↑cc_task,i+max_transport)
∧ @(↓put_transmit,i+max_transport-1) ≤ @(↑cc_task,i+max_transport)
∧ @(↑free_compress,i) > @(↑cc_task,i+max_transport)
```

which is the l.h.s. of Axiom 5.28. Thus, by Axiom 5.28 there exists a k such that equation (5.28) holds.

```
   @(↑free_compress,k) > @(↑cc_task,i+max_transport)
 ∧ @(↓free_compress,k) < @(↓cc_task,i+max_transport)              (5.28)
```

I now show that $k \geq i$. There are two cases, depending on i.

<u>Case 2a</u>: Assume $i = 1$. Then $k \geq i$.

<u>Case 2b</u>: Assume $i > 1$.

By equation (5.28), Axiom 5.2, and the l.h.s. of the theorem, and Axiom 5.1

```
 @(↑free_compress,k) > @(↑cc_task,i+max_transport)
                     > @(↓cc_task,i+max_transport-1)
                     > @(↓free_compress,i-1)
                     > @(↑free_compress,i-1)
```

Thus, by the contrapositive of Theorem 5.3, $k > i - 1$.

Thus in both case 2a and 2b, $k \geq i$. By Theorem 5.3 and equation (5.28)

```
 @(↓free_compress,i) ≤ @(↓free_compress,k)
                     ≤ @(↓cc_task,i+max_transport)
```

This proves the theorem. □

5.7.5 The Main Theorem

I am now ready to develop the proof of the main theorem of the chapter. As described previously, the proof is an induction that uses the six theorems developed above as the central steps in showing that the six groups of conjuncts in the theorem hold.

Theorem 5.40

```
   @(↑alloc_digitize,i) > @(↑vbi1_part2,i)
 ∧ @(↓alloc_digitize,i) < @(↓vbi1_part2,i)
 ∧ @(↑digitize,i) > @(↑vbi1_part2,i)
 ∧ @(↓digitize,i) < @(↓vbi1_part2,i)
 ∧ @(↑put_next_digitizing,i) > @(↑vbi1_part2,i)
 ∧ @(↓put_next_digitizing,i) < @(↓vbi1_part2,i)
 ∧ frame(put_next_digitizing,i) = frame(digitize,i)
```

```
∧ @(↑get_next_digitizing,i) > @(↑vbi0_task,i)
∧ @(↓get_next_digitizing,i) < @(↓vbi0_task,i)
∧ @(↑put_digitizing,i) > @(↑vbi0_task,i)
∧ @(↓put_digitizing,i) < @(↓vbi0_task,i)
∧ frame(put_digitizing,i) = frame(get_next_digitizing,i)

∧ @(↑get_digitizing,i) > @(↑vbi0_task,i+1)
∧ @(↓get_digitizing,i) < @(↓vbi0_task,i+1)
∧ @(↑put_compress_source,i) > @(↑vbi0_task,i+1)
∧ @(↓put_compress_source,i) < @(↓vbi0_task,i+1)
∧ frame(put_compress_source,i) = frame(get_digitizing,i)
∧ @(↑alloc_compress,i) > @(↑vbi0_task,i+1)
∧ @(↓alloc_compress,i) < @(↓vbi0_task,i+1)
∧ @(↑put_compress_sink,i) > @(↑vbi0_task,i+1)
∧ @(↓put_compress_sink,i) < @(↓vbi0_task,i+1)
∧ frame(put_compress_sink,i) = frame(get_digitizing,i)
∧ @(↑compress,i) > @(↑vbi0_task,i+1)
∧ @(↓compress,i) < @(↓vbi0_task,i+1)
∧ frame(compress,i) = frame(get_digitizing,i)

∧ @(↑get_compress_sink,i) > @(↑cc_task,i)
∧ @(↓get_compress_sink,i) < @(↓cc_task,i)
∧ @(↑put_transmit,i) > @(↑cc_task,i)
∧ @(↓put_transmit,i) < @(↓cc_task,i)
∧ frame(put_transmit,i) = frame(get_compress_sink,i)

∧ [   i ≤ digitize_buffers
    ∨ @(↓free_digitize,i-digitize_buffers) ≤ @(↑vbi1_part2,i)]

∧ [   i ≤ max_transport
    ∨ @(↓free_compress,i-max_transport) ≤ @(↓cc_task,i)]
```

Proof: By induction. Assume that the proposition holds $\forall i < N$. I will show that it holds for N. There are six steps in the proof. In the six steps, I will show that the six equations (5.29), (5.34), (5.36), (5.39), (5.43), and (5.46) can be derived either from the induction hypothesis or from previous steps in the proof.

Step 1: I begin by showing that equation (5.29) holds.

```
  N ≤ digitize_buffers
∨ @(↓free_digitize,N-digitize_buffers) ≤ @(↑vbi1_part2,N)  (5.29)
```

There are two cases to be considered, depending on N.

<u>Case 1</u>: Assume $N \le digitize_buffers$. Equation (5.29) holds trivially.

Case 2: Assume $N > digitize_buffers$. Let $m = N\text{-}digitize_buffers$. Thus, equation (5.29) will hold only if equation (5.30) holds.

$$@(\downarrow free_digitize,m) \le @(\uparrow vbi1_part2,N) \tag{5.30}$$

There are four substeps required to show that equation (5.30) holds. In steps 1a-1c, I show that equations (5.31), (5.32), and (5.33) hold. In step 1d, I use Theorem 5.35 to show that equation (5.30) holds.

Step 1a: I begin by showing that equation (5.31) holds.

$$@(\downarrow put_compress_source,m) \le @(\uparrow vbi1_part1,N) \tag{5.31}$$

By the induction hypothesis, Theorem 5.30, the bound on *digitize_buffers* given in Figure 5-2, Theorem 5.34, and Axiom 5.12

$$\begin{aligned} @(\downarrow put_compress_source,m) &\le @(\downarrow vbi0_task,m+1) \\ &\le 2(m+1)\cdot \mathit{vbi_period} \\ &\le 2(N-\mathit{digitize_buffers}+1)\cdot \mathit{vbi_period} \\ &\le 2(N-2)\cdot \mathit{vbi_period} \\ &\le (2N-2)\cdot \mathit{vbi_period} \\ &\le @(\uparrow vbi1_task,N) \\ &\le @(\uparrow vbi1_part1,N) \end{aligned}$$

Thus equation (5.31) holds.

Step 1b: Next I show that equation (5.32) holds.

$$@(\Omega CC,m) \le @(\uparrow vbi1_part1,N) \tag{5.32}$$

By Axiom 5.8, the induction hypothesis, Theorem 5.30, the bound on *compress_request* given in Figure 5-2, the bound on *digitize_buffers* given in Figure 5-2, Theorem 5.34, and Axiom 5.12

$$\begin{aligned} @(\Omega CC,m) &\le @(\downarrow compress,m) + \mathit{compress_request} \\ &\le @(\downarrow vbi0_task,m+1) + \mathit{compress_request} \\ &\le 2(m+1)\cdot \mathit{vbi_period} + \mathit{compress_request} \\ &\le 2(m+1)\cdot \mathit{vbi_period} + 2\cdot \mathit{vbi_period} \\ &\le (2m+4)\cdot \mathit{vbi_period} \\ &\le (2(N-\mathit{digitize_buffers})+4)\cdot \mathit{vbi_period} \\ &\le (2(N-3)+4)\cdot \mathit{vbi_period} \\ &\le (2N-2)\cdot \mathit{vbi_period} \\ &\le @(\uparrow vbi1_task,N) \\ &\le @(\uparrow vbi1_part1,N) \end{aligned}$$

Thus equation (5.32) holds.

Step 1c: Next I show that equation (5.33) holds.

```
    m = 1
  ∨ @(↓free_digitize,m-1) < @(↓vbi1_part1,N-1)
                                                                    (5.33)
```

If $m = 1$, then equation (5.33) holds trivially. If $m > 1$, then equation (5.33) holds by the induction hypothesis. In either case, equation (5.33) holds.

Step 1d: Together, equations (5.31), (5.32), and (5.33) are the l.h.s. of Theorem 5.35. Applying the theorem for $i = m$

```
  @(↓free_digitize,m) ≤ @(↓vbi1_part1,N)
```

and thus by Axiom 5.12

```
  @(↓free_digitize,m) ≤ @(↑vbi1_part2,N)
```

Thus equation (5.30) holds.

End of Step 1: In both cases, equation (5.29) holds.

Step 2: Next, I show that equation (5.34) holds.

```
    @(↑alloc_digitize,N) > @(↑vbi1_part2,N)
  ∧ @(↓alloc_digitize,N) < @(↓vbi1_part2,N)
  ∧ @(↑digitize,N) > @(↑vbi1_part2,N)
  ∧ @(↓digitize,N) < @(↓vbi1_part2,N)
  ∧ @(↑put_next_digitizing,N) > @(↑vbi1_part2,N)
  ∧ @(↓put_next_digitizing,N) < @(↓vbi1_part2,N)
  ∧ frame(put_next_digitizing,N) = frame(digitize,N)         (5.34)
```

There are two substeps required to show that equation (5.34) holds. In step 2a, I show that equation (5.35) holds. In step 2b, I use Theorem 5.36 to show that equation (5.34) holds.

Step 2a: I begin by showing that equation (5.35) holds.

```
    N = 1
  ∨ [  @(↑alloc_digitize,N-1) > @(↑vbi1_part2,N-1)
     ∧ @(↓alloc_digitize,N-1) < @(↓vbi1_part2,N-1)]          (5.35)
```

If $N = 1$, then equation (5.35) holds trivially. If $N > 1$, then equation (5.35) holds by the induction hypothesis. In either case, equation (5.35) holds.

Step 2b: Together with equation (5.29), equation (5.35) forms the l.h.s. of Theorem 5.36. Applying the theorem for $i = N$ gives equation (5.34).

End of Step 2: Thus equation (5.34) holds.

Step 3: Next, I show that equation (5.36) holds.

```
  @(↑get_next_digitizing,N) > @(↑vbi0_task,N)
∧ @(↓get_next_digitizing,N) < @(↓vbi0_task,N)
∧ @(↑put_digitizing,N) > @(↑vbi0_task,N)
∧ @(↓put_digitizing,N) < @(↓vbi0_task,N)
∧ frame(put_digitizing,N) = frame(get_next_digitizing,N)   (5.36)
```

There are three substeps required to show that equation (5.36) holds. In steps 3a and 3b, I show that equations (5.37) and (5.38) hold. In step 3c, I use Theorem 5.31 to show that equation (5.36) holds.

Step 3a: I begin by showing that equation (5.37) holds.

```
@(↓put_next_digitizing,N) ≤ @(↑vbi0_task,N)                  (5.37)
```

By equation (5.34), Axiom 5.12, and Theorems 5.34 and 5.30

```
@(↓put_next_digitizing,N) < @(↓vbi1_part2,N)
                          < @(↓vbi1_task,N)
                          < (2N-1)·vbi_period
                          < @(↑vbi0_task,N)
```

Thus equation (5.37) holds.

Step 3b: Next I show that equation (5.38) holds.

```
  N = 1
∨ [  @(↑get_next_digitizing,N-1) > @(↑vbi0_task,N-1)
   ∧ @(↓get_next_digitizing,N-1) < @(↓vbi0_task,N-1)]]     (5.38)
```

If $N = 1$, then equation (5.38) holds trivially. If $N > 1$, then equation (5.38) holds by the induction hypothesis. In either case, equation (5.38) holds.

Step 3c: Together, equations (5.37) and (5.38) are the l.h.s. of Theorem 5.31. Applying the theorem for $i = N$ gives equation (5.36).

End of Step 3: Thus equation (5.36) holds.

Step 4: Next, I show that equation (5.39) holds.

```
  @(↑get_digitizing,N) > @(↑vbi0_task,N+1)
∧ @(↓get_digitizing,N) < @(↓vbi0_task,N+1)
∧ @(↑put_compress_source,N) > @(↑vbi0_task,N+1)
∧ @(↓put_compress_source,N) < @(↓vbi0_task,N+1)
∧ frame(put_compress_source,N) = frame(get_digitizing,N)
∧ @(↑alloc_compress,N) > @(↑vbi0_task,N+1)
∧ @(↓alloc_compress,N) < @(↓vbi0_task,N+1)
∧ @(↑put_compress_sink,N) > @(↑vbi0_task,N+1)
∧ @(↓put_compress_sink,N) < @(↓vbi0_task,N+1)
∧ frame(put_compress_sink,N) = frame(get_digitizing,N)
∧ @(↑compress,N) > @(↑vbi0_task,N+1)
∧ @(↓compress,N) < @(↓vbi0_task,N+1)
∧ frame(compress,N) = frame(get_digitizing,N)             (5.39)
```

There are four substeps required to show that equation (5.39) holds. In steps 4a-4c, I show that equations (5.40), (5.41) and (5.42) hold. In step 4d, I use Theorem 5.33 to show that equation (5.39) holds.

Step 4a: I begin by showing that equation (5.40) holds.

```
  N = 1
∨ [  @(↑get_digitizing,N-1) > @(↑vbi0_task,N)
   ∧ @(↓get_digitizing,N-1) < @(↓vbi0_task,N)]            (5.40)
```

If $N = 1$, then equation (5.40) holds trivially. If $N > 1$, then equation (5.40) holds by the induction hypothesis. In either case, equation (5.40) holds.

Step 4b: Next, I show that equation (5.41) holds.

```
  N ≤ compress_buffers
∨ @(↓free_compress,N-compress_buffers) ≤ @(↑vbi0_task,N+1) (5.41)
```

There are two cases, depending on N.

Case 1: Assume $N \leq$ *compress_buffers*. Then equation (5.41) holds trivially.

Case 2: Assume $N >$ *compress_buffers*. Let $m = N$-*compress_buffers*.

By the bound on *compress_buffers* given in Figure 5-2, the induction hypothesis, Theorem 5.37, the induction hypothesis, and two uses of Theorem 5.30

```
@(↓free_compress,m) ≤ @(↓free_compress,N-compress_buffers)
                     ≤ @(↓free_compress,N-max_transport-2)
                     ≤ @(↓cc_task,N-2)
                     ≤ @(↓compress,N-2) + 3·vbi_period
                     ≤ @(↓vbi0_task,N-1) + 3·vbi_period
                     ≤ 2(N-1)·vbi_period + 3·vbi_period
                     ≤ (2N+1)·vbi_period
                     ≤ (2(N+1)-1)·vbi_period
                     ≤ @(↑vbi0_task,N+1)
```

Thus in either case equation (5.41) holds.

Step 4c: Next, I show that equation (5.42) holds.

```
  N = 1
∨ [  @(↑alloc_compress,N-1) > @(↑vbi0_task,N)
   ∧ @(↓alloc_compress,N-1) < @(↓vbi0_task,N)]             (5.42)
```

If $N = 1$, then equation (5.42) holds trivially. If $N > 1$, then equation (5.42) holds by the induction hypothesis. In either case, equation (5.42) holds.

Step 4d: Along with equation (5.36), equations (5.40), (5.41) and (5.42) form the l.h.s. of Theorem 5.33. Applying the theorem for $i = N$ gives equation (5.39).

End of Step 4: Thus equation (5.39) holds.

Step 5: Next, I show that equation (5.43) holds.

```
  @(↑get_compress_sink,N) > @(↑cc_task,N)
∧ @(↓get_compress_sink,N) < @(↓cc_task,N)
∧ @(↑put_transmit,N) > @(↑cc_task,N)
∧ @(↓put_transmit,N) < @(↓cc_task,N)
∧ frame(put_transmit,N) = frame(get_compress_sink,N)      (5.43)
```

There are three substeps required to show that equation (5.43) holds. In steps 5a and 5b, I show that equations (5.44) and (5.45) hold. In step 5c, I use Theorem 5.38 to show that equation (5.43) holds.

Step 5a: I begin by showing that equation (5.44) holds.

```
@(↓put_compress_sink,N) ≤ @(↑cc_task,N)                    (5.44)
```

By Theorem 5.37, Axiom 5.1 and equation (5.39)

```
@(↑cc_task,N) ≥ @(↓compress,N)
              > @(↑compress,N)
              > @(↑vbi0_task,N+1)
```

and thus by Axiom 5.13

```
@(↑cc_task,N) > @(↓vbi0_task,N+1)
```

Combining this with equation (5.39) yields

```
@(↓put_compress_sink,N) < @(↓vbi0_task,N+1)
                        ≤ @(↑cc_task,N)
```

Thus equation (5.44) holds.

Step 5b: Next I shown that equation (5.45) holds.

```
  N = 1
∨ [  @(↑get_compress_sink,N-1) > @(↑cc_task,N-1)
   ∧ @(↓get_compress_sink,N-1) < @(↓cc_task,N-1)]          (5.45)
```

If $N = 1$, then equation (5.45) holds trivially. If $N > 1$, then equation (5.45) holds by the induction hypothesis. In either case, equation (5.45) holds.

Step 5c: Together, equations (5.44) and (5.45) form the l.h.s. of Theorem 5.38. Applying the theorem for $i = N$ gives equation (5.43).

End of Step 5: Thus equation (5.43) holds.

Step 6: Next, I show that equation (5.46) holds.

```
  N ≤ max_transport
∨ @(↓free_compress,N-max_transport) ≤ @(↓cc_task,N)        (5.46)
```

There are two cases to be considered, depending on *N*.

Case 1: Assume $N \leq max_transport$. Equation (5.46) holds trivially.

Case 2: Assume $N > max_transport$. Let $m = N\text{-}max_transport$. Thus, equation (5.46) will hold only if equation (5.47) holds.

```
@(↓free_compress,m) ≤ @(↓cc_task,N)                        (5.47)
```

There are three substeps required to show that equation (5.47) holds. In steps 6a and 6b, I show that equations (5.48) and (5.49) hold. In step 6c, I use Theorem 5.39 to show that equation (5.47) holds.

Step 6a: I begin by showing that equation (5.48) holds.

```
@(↓put_transmit,m+max_transport-1) ≤ @(↑cc_task,m+max_transport)
                                                            (5.48)
```

By the induction hypothesis, Axiom 5.2, and the definition of *m*

```
@(↓put_transmit,m+max_transport-1) ≤ @(↓cc_task,N-1)
                                   ≤ @(↑cc_task,N)
                                   ≤ @(↑cc_task,m+max_transport)
```

Thus equation (5.48) holds.

Step 6b: Next I show that equation (5.49) holds.

```
  m = 1
∨ @(↓free_compress,m-1) ≤ @(↓cc_task,m+max_transport-1)    (5.49)
```

If $m = 1$, then equation (5.49) holds trivially. If $m > 1$, then by the induction hypothesis, Axiom 5.1, and the definition of *m*

```
@(↓free_compress,m-1) ≤ @(↓cc_task,N-1)
                      ≤ @(↑cc_task,N-1)
                      ≤ @(↑cc_task,m+max_transport-1)
```

Thus equation (5.49) holds.

Step 6c: Together with equation (5.43), equations (5.48) and (5.49) form the l.h.s. of Theorem 5.39. Applying the theorem for $i = m$ gives equation (5.47). Thus equation (5.46) holds.

End of Step 6: Thus in either case equation (5.46) holds.

Together (5.29), (5.34), (5.36), (5.39), (5.43), and (5.46) prove the theorem. □

5.7.6 Proof of the Correctness Condition

I am now ready to develop the proof of the correctness condition given in Figure 5-10. Before proving the theorem, I prove two lemmas: the first gives the interval within which

the ith `digitize` action occurs in terms of VBI interrupts and the second shows that for several actions, the ith instance of the action processes the frame with frame number i.

Lemma 5.41

```
  @(↑digitize,i) > @(ΩVBI,2i-1)
∧ @(↓digitize,i) < @(ΩVBI,2i)
```

Proof: By Theorem 5.40, Axiom 5.12, Theorem 5.34, and Axiom 5.7

```
@(↑digitize,i) > @(↑vbi1_part2,i)
               > @(↑vbi1_task,i)
               > (2i-2)·vbi_period
               > @(ΩVBI,2i-1)
```

Similarly, by Theorem 5.40, Axiom 5.12, Theorem 5.34, and Axiom 5.7

```
@(↓digitize,i) < @(↓vbi1_part2,i)
               < @(↓vbi1_task,i)
               < (2i-1)·vbi_period
               < @(ΩVBI,2i)
```

Together, these prove the lemma. □

Lemma 5.42

```
  index(digitize,i) = i
∧ index(compress,i) = i
∧ index(put_transmit,i) = i
```

Proof: By Lemma 5.41

```
  @(↑digitize,i) > @(ΩVBI,2i-1)
∧ @(↓digitize,i) < @(ΩVBI,2i)
```

and by Axiom 5.1

```
@(↓digitize,i) > @(↑digitize,i)
```

so it is the case that

```
  @(↓digitize,i) > @(ΩVBI,2i-1)
∧ @(↓digitize,i) < @(ΩVBI,2i)
```

and thus by Axiom 5.4

```
frame(digitize,i) = i                                    (5.50)
```

By Theorem 5.40

```
  frame(put_next_digitizing,i) = frame(digitize,i)
∧ frame(put_digitizing,i) = frame(get_next_digitizing,i)
∧ frame(compress,i) = frame(get_digitizing,i)
∧ frame(put_compress_sink,i) = frame(get_digitizing,i)
∧ frame(put_transmit,i) = frame(get_compress_sink,i)          (5.51)
```

and by Axiom 5.5

```
  frame(get_next_digitizing,i) = frame(put_next_digitizing,i)
∧ frame(get_digitizing,i) = frame(put_digitizing,i)
∧ frame(get_compress_sink,i) = frame(put_compress_sink,i)    (5.52)
```

Combining equations (5.50), (5.51), and (5.52) yields

```
frame(digitize,i) = frame(put_next_digitizing,i)
                  = frame(get_next_digitizing,i)
                  = frame(put_digitizing,i)
                  = frame(get_digitizing,i)
                  = frame(compress,i)
                  = frame(put_compress_sink,i)
                  = frame(get_compress_sink,i)
                  = frame(put_transmit,i)
                  = i
```

Thus

```
  frame(digitize,i) = i
∧ frame(compress,i) = i
∧ frame(put_transmit,i) = i
```

and by Axiom 5.6

```
  index(digitize,i) = i
∧ index(compress,i) = i
∧ index(put_transmit,i) = i
```

This proves the lemma. □

Theorem 5.43

```
  @(↑digitize,index(digitize,i)) > @(ΩVBI,2i-1)
∧ @(↓digitize,index(digitize,i)) ≤ @(ΩVBI,2i)
∧  ∃j [  @(↑digitize,j) > @(ΩVBI,2i+1)
       ∧ @(↓digitize,j) ≤ @(ΩVBI,2i+2)]
∧ ~∃j [  j≠index(digitize,i)
       ∧ @(↑digitize,j) > @(ΩVBI,2i-1)
       ∧ @(↓digitize,j) ≤ @(ΩVBI,2i+1)
      ]
∧ @(↑compress,index(compress,i)) ≥ @(ΩVBI,2i+2)
∧ @(↑put_transmit,index(put_transmit,i)) ≥ @(ΩCC,index(compress,i))
∧ @(↓put_transmit,index(put_transmit,i)) - @(ΩVBI,2i) ≤ 6·vbi_period
```

Proof:

Step 1: I begin by showing that equation (5.53) holds.

```
  @(↑digitize,index(digitize,i)) > @(ΩVBI,2i-1)
∧ @(↓digitize,index(digitize,i)) ≤ @(ΩVBI,2i)                  (5.53)
```

By Lemma 5.41

```
  @(↑digitize,i) > @(ΩVBI,2i-1)
∧ @(↓digitize,i) < @(ΩVBI,2i)                                  (5.54)
```

By Lemma 5.42

```
index(digitize,i) = i
```

Substituting *index*(digitize,i) for *i* in equation (5.54) yields equation (5.53).

Step 2: Next I show that equation (5.55) holds.

```
∃j [  @(↑digitize,j) > @(ΩVBI,2i+1)
    ∧ @(↓digitize,j) ≤ @(ΩVBI,2i+2)]                           (5.55)
```

By Lemma 5.41

```
  @(↑digitize,i+1) > @(ΩVBI,2i+1)
∧ @(↓digitize,i+1) < @(ΩVBI,2i+2)
```

Thus equation (5.55) holds.

Step 3: Next I show that equation (5.56) holds.

```
~∃j [   j≠index(digitize,i)
      ∧ @(↑digitize,j) > @(ΩVBI,2i-1)
      ∧ @(↓digitize,j) ≤ @(ΩVBI,2i+1)
     ]                                                          (5.56)
```

This is equivalent to equation (5.57), so equation (5.56) holds if equation (5.57) holds.

```
∀j [   j = index(digitize,i)
     ∨ @(↑digitize,j) ≤ @(ΩVBI,2i-1)
     ∨ @(↓digitize,j) > @(ΩVBI,2i+1)
    ]                                                           (5.57)
```

To show that equation (5.57) holds, there are three cases depending on j.

Case 1: Assume $j < i$.

By Axiom 5.1, Lemma 5.41, Axiom 5.7, the assumption about j, and Axiom 5.7

```
@(↑digitize,j) ≤ @(↓digitize,j)
               ≤ @(ΩVBI,2j)
               ≤ (2j-1)·vbi_period
               ≤ (2(i-1)-1)·vbi_period
               ≤ (2i-2)·vbi_period
               ≤ @(ΩVBI,2i-1)
```

Thus equation (5.57) holds.

Case 2: Assume $j > i$.

By Axiom 5.1, Lemma 5.41, Axiom 5.7, the assumption about j, and Axiom 5.7

```
@(↓digitize,j) > @(↑digitize,j)
               > @(ΩVBI,2j-1)
               > (2j-2)·vbi_period
               > (2(i+1)-2)·vbi_period
               > (2i)·vbi_period
               > @(ΩVBI,2i+1)
```

Thus equation (5.57) holds.

Case 3: Assume $j = i$. By Lemma 5.42

```
j = index(digitize,i)
```

Thus equation (5.57) holds.

In all cases, equation (5.57) holds, so equation (5.56) holds.

Step 4: Next I show that equation (5.58) holds.

$$@(\uparrow\texttt{compress},\mathit{index}(\texttt{compress},\texttt{i})) > @(\Omega\texttt{VBI},\texttt{2i+2}) \tag{5.58}$$

By Theorems 5.40 and 5.30, and Axiom 5.7

$$\begin{aligned} @(\uparrow\texttt{compress},\texttt{i}) &> @(\uparrow\texttt{vbi0_task},\texttt{i+1}) \\ &> (\texttt{2i+1})\cdot \mathit{vbi_period} \\ &> @(\Omega\texttt{VBI},\texttt{2i+2}) \end{aligned} \tag{5.59}$$

By Lemma 5.42

$$\mathit{index}(\texttt{compress},\texttt{i}) = \texttt{i}$$

Substituting $\mathit{index}(\texttt{compress},\texttt{i})$ for i in equation (5.59) yields equation (5.58).

Step 5: Next I show that equation (5.60) holds.

$$@(\uparrow\texttt{put_transmit},\mathit{index}(\texttt{put_transmit},\texttt{i})) > @(\Omega\texttt{CC},\mathit{index}(\texttt{compress},\texttt{i})) \tag{5.60}$$

By Theorem 5.40 and Axioms 5.11, 5.1, and 5.10

$$\begin{aligned} @(\uparrow\texttt{put_transmit},\texttt{i}) &> @(\uparrow\texttt{cc_task},\texttt{i}) \\ &> @(\downarrow\texttt{send_cc},\texttt{i}) \\ &> @(\uparrow\texttt{send_cc},\texttt{i}) \\ &> @(\Omega\texttt{CC},\texttt{i}) \end{aligned} \tag{5.61}$$

and by Lemma 5.42

$$\mathit{index}(\texttt{put_transmit},\texttt{i}) = \texttt{i}$$

Substituting $\mathit{index}(\texttt{put_transmit},\texttt{i})$ for i in equation (5.61) yields equation (5.60)

Step 6: Finally, I show that equation (5.62) holds.

$$@(\downarrow\texttt{put_transmit},\mathit{index}(\texttt{put_transmit},\texttt{i})) - @(\Omega\texttt{VBI},\texttt{2i}) \leq 6\cdot\mathit{vbi_period} \tag{5.62}$$

By Theorems 5.40, 5.37, 5.40, and 5.30

$$
\begin{aligned}
@(\downarrow\texttt{put_transmit},i) &< @(\downarrow\texttt{cc_task},i) \\
&< @(\downarrow\texttt{compress},i) + 3\cdot\mathit{vbi_period} \\
&< @(\downarrow\texttt{vbi0_task},i+1) + 3\cdot\mathit{vbi_period} \\
&< 2(i+1)\cdot\mathit{vbi_period} + 3\cdot\mathit{vbi_period} \\
&< (2i+5)\cdot\mathit{vbi_period}
\end{aligned}
$$

and thus by Axiom 5.7

$$
\begin{aligned}
@(\downarrow\texttt{put_transmit},i) - @(\Omega\texttt{VBI},2i) &< (2i+5)\cdot\mathit{vbi_period} - (2i-1)\cdot\mathit{vbi_period} \\
&< 6\cdot\mathit{vbi_period}
\end{aligned}
$$

Thus equation (5.62) holds.

Together (5.53), (5.55), (5.56), (5.58), (5.60), and (5.62) show the theorem. □

5.8 A Note on the Lower Bound

Previously in this chapter, I have argued that 100 ms. is an upper bound on the time required for a video frame to be correctly acquired, compressed, and delivered to the network. Recall that delay jitter can be reduced or eliminated simply by buffering the frames to account for the difference between the actual delay experienced by each frame and this upper bound. However, unless the upper bound is reasonably tight, such a strategy would lead to artificially high delay. Thus, it is useful to briefly consider the lower bound.

Recall that the delay experienced by a video frame on the acquisition-side is defined as the elapsed time between the VBI logical interrupt that occurs at the start of the digitization of the frame and the time the frame is placed on the `transmit` queue. Between these two events, the frame is digitized and compressed. Digitization by the ActionMedia hardware always takes 33 ms. Compression by the ActionMedia hardware takes between 22 and 28 ms. Thus, the frame experiences acquisition-side delay of at least 55 ms. simply due to hardware processing. As a result, 55 ms. is an extremely conservative estimate of the lower bound on acquisition-side delay. Effectively, this lower bound holds even under assumptions that software operations take no time, and that work is always performed as soon as possible. Under more realistic assumptions a tighter lower bound could be determined. However, given the delay jitter experienced by frames when transmitted over the networks used in this work, even this conservative lower bound is sufficient to show that the delay jitter experienced can be reduced or eliminated without markedly increasing the end-to-end delay jitter.

5.9 Discussion

In this chapter, I have presented an axiomatic specification of that portion of the acquisition-side of the application that is responsible for acquiring, digitizing, and compressing video frames. I then used this specification to reason formally about properties of the acquisition-side. In particular, I showed that each frame that is generated by the ActionMedia hardware is correctly acquired, compressed, and delivered to the network, and that the time required to do so is at most 100 ms. By proving that the acquisition-side side delay experienced by video frames is bounded, I have demonstrated that it is feasible to reduce or eliminate the delay jitter experienced by video frames on the acquisition-side. Furthermore, since 55 ms. is a conservative estimate of the lower bound on delay, I have demonstrated that the delay jitter experienced by video frames on the acquisition-side can be reduced or eliminated without introducing artificially high delays.

By taking advantage of the deadline and mutual exclusion properties that were shown in Chapter 4, I have simplified the analysis presented here by eliminating the need to reason about detailed interactions between tasks under all possible orderings of events; the effect of executing a task invocation depended only on the state of the system at the time the task invocation started execution. Thus, the effect of executing a single task could be modeled without reference to other tasks.

More importantly, I have enforced a separation of concerns. The only assumption included in the axiomatic specification about the times at which task invocations execute is that tasks execute after they are invoked and prior to their deadline. Thus, the argument that delay is bounded and that every frame is acquired and displayed correctly is free of detailed assumptions about how long tasks and actions require to execute, assumptions about scheduling, and assumptions about the existence of other tasks in the application. As a result, code can be added to tasks (*i.e.*, changing the cost) and other tasks can be added to the application without affecting the logical argument presented here.

Having shown that the delay experienced by video frames on the acquisition-side is bounded, it is straightforward to extend the analysis to show that the delay experienced by audio frames on the acquisition-side is also bounded. All that is necessary is to extend the axiomatic specification of the acquisition-side to include axioms representing the behavior of the tasks that process audio; the proof that audio frames are delivered to the network in bounded time is analogous to that for video frames.

I can also extend the analysis to show that the delays experienced by audio and video frames on the display-side are bounded. In general, the same approach can be used: represent the display-side in terms of the formal model, use the feasibility test to show that tasks execute prior to the application-defined deadlines, develop an axiomatic specification, and derive the bounded delay property. However, there is one additional difficulty that must be addressed: how can the task that executes whenever a new packet arrives (*i.e.* the "`receive_complete`" task) be represented in the abstract model? There are two problems. First, since packets may be sent to the display workstation from many sources, an arbitrary number of packets could potentially arrive in any given interval. Furthermore, even if we consider only packets sent by the acquisition-side of the application, since the delays experienced by packets in the network are variable, it is still possible for an arbitrary number of packets to arrive in any given interval. One solution to this problem is to change the implementation; rather than execute a task to process a new packet whenever it arrives, execute a task periodically to process any packets that have arrived in the most recent period (*i.e.*, a polling implementation instead an interrupt-driven implementation).

Overall, in these three chapters, I have argued that through the use of real-time systems design, analysis, and implementation techniques, that it is possible to control the delay jitter experienced by continuous media frames due to causes other than transmission over the network. In the next two chapters, I will address the question of ameliorating the effect of the delay jitter than cannot be controlled.

Chapter VI
Policies for Managing Delay Jitter

6.1 Introduction

In the previous chapters, I have demonstrated that through the use of real-time systems design, analysis, and implementation techniques, it is possible to bound the delay jitter experienced by continuous media frames due to causes other than transmission over the network. In this and the following chapter, I address the question of displaying CM frames in the presence of the potentially unbounded delay jitter incurred when transmitting over the network.

In Chapter 1, I discussed the fact that, in the presence of delay jitter, there is a fundamental tradeoff between display latency and gap frequency; the lower the display latency, the higher the probability of encountering an end-to-end delay sufficient to cause a gap. An application that displays continuous media frames must manage this tradeoff to produce a balance between display latency and gaps that results in good quality playout.

It is useful to consider the tradeoff between display latency and gap frequency in the context of the idealized application for acquiring, processing and displaying frames of live continuous media that was illustrated in Figure 1-1. The design of this application is a distributed pipeline that includes a set of buffers placed immediately before the display stage called the display queue. The tradeoff between display latency and gap frequency can be viewed as a tradeoff between a long display queue and a short display queue. If the display queue contains many frames, then a gap will occur only if a frame incurs a very long end-to-end delay; however the long display queue implies that frames are played with high display latency. If the display queue contains few frames, then a much shorter end-to-end delay may cause a gap, but frames are played with lower display latency.

A policy for managing the display queue can be defined as a policy for choosing whether or not newly arrived frames are inserted into the queue, when frames may be removed from the queue to be played, and if and when frames are discarded from the queue without

being played. In effect, a policy for managing the display queue can be viewed as a policy for managing the tradeoff between display latency and gap frequency. I will refer to such policies as delay jitter management policies.

In this chapter, I describe three delay jitter management policies. Two policies, the I-policy and the E-policy are taken from the literature; the third, *queue monitoring*, is a new policy that I have developed. In Chapter 7, I evaluate the performance of these policies in an empirical study using the workstation-based videoconferencing application described in Chapter 2.

Section 6.2 describes the effect of delay jitter on the display of continuous media, illustrates the basic principles of managing the tradeoff between display latency and gap frequency, and defines the I- and E-policies. Section 6.3 presents the queue monitoring policy.

6.2 Effect of Delay Jitter

In order to sustain continuous playout without any gaps, an application must play every frame with a fixed display latency that is greater than the worst-case end-to-end delay that will be encountered during a conference. There are two difficulties with this approach. First, when frames are transmitted over the networks considered in this work, the worst-case delay may not be known. Second, it is not clear that the primary goal should be playout with no gaps. Display latency and gaps are only some of the important factors in determining the perceived quality of the playout [21]. It is likely that in many applications, as long as gaps occur infrequently, playout with low latency and some gaps will be preferable to playout with high latency and no gaps. Therefore, if an application always plays frames with a display latency greater than the worst-case delay and if the worst-case delay is rarely observed in practice, then most frames will be displayed with latency higher than necessary to support good quality playout.

If the worst-case delay is not known, or if an application chooses to play frames with a display latency less than the worst-case end-to-end delay, then gaps in the playout may occur. If so, then the application must address two issues. First, gaps occur when there is no new frame available to be played; what should the application play instead? The workstation-based video conferencing application described in Chapter 2 uses a simple strategy: gaps in the video stream are covered by replaying the previous frame and gaps in the audio stream are covered by playing silence.

The second issue that must be addressed when gaps are possible is the question of what should be done with a frame whose late arrival resulted in a gap? There are two choices: either the late frame can be discarded or it can be displayed. These choices define two delay jitter management policies that Naylor and Kleinrock call the *I-Policy* and the *E-Policy* [37]. Under the I-policy, all frames are displayed at a fixed display latency; each frame that arrives with an end-to-end delay greater than this latency is discarded. The particular display latency is a parameter of the policy. Under the E-policy, the late frame is displayed.

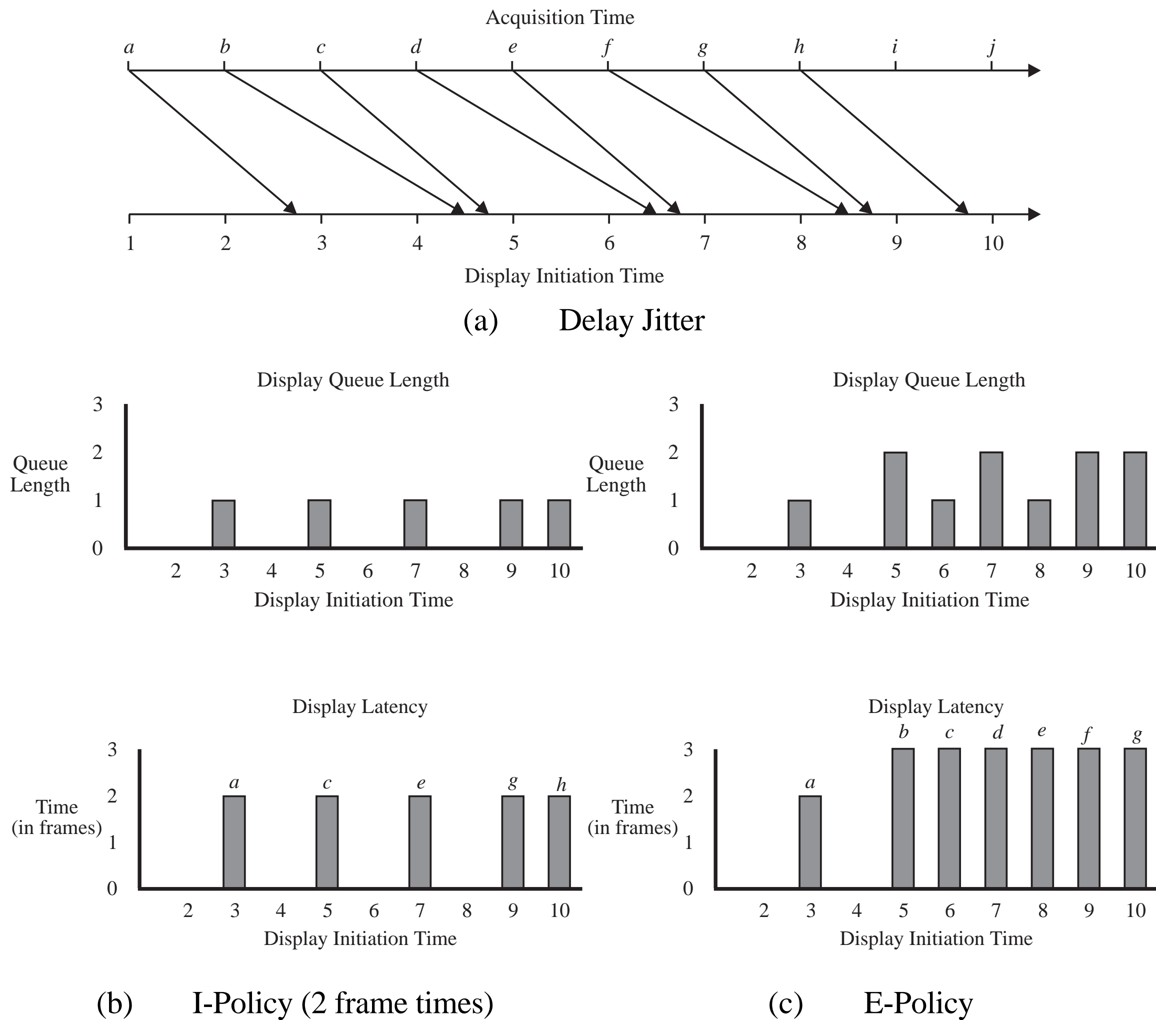

Figure 6-1: I-Policy and E-Policy with Persistent Delay Jitter

Figure 6-1 illustrates the behavior of the I- and E-policies in response to late frames. Figure 6-1a shows the acquisition and display times for eight frames. Tick marks on the upper timeline indicate *acquisition times*, the times at which new frames are acquired.

Tick marks on the lower timeline indicate *display initiation times*, the times at which the new frames are displayed. (In the application described in Chapter 2, the acquisition time of a video frame is defined as the time at which the Vbi0 logical interrupt that defines the start of the frame occurs on the acquisition-side of the application; display initiation times are defined by the times at which the Vbi0 interrupts that result in the display of new video frames occur on the display-side.) Each diagonal arrow represents the end-to-end delay of an individual frame, extending from the time at which it was acquired to the time it is placed in the display queue. Throughout these examples, the acquisition time of a frame (*i.e.*, points *a*, *b*, *c*, etc.) is used to refer to individual frames.

Figure 6-1b shows the effect of executing the I-Policy on a sequence of frames arriving at the display queue with the end-to-end delays shown in Figure 6-1a. In this example, the display latency parameter of the I-Policy is two frame times. (For simplicity in the examples, time is represented as multiples of the time to acquire or display a frame). Each frame that arrives with an end-to-end delay less than two frame times is held in the display queue until it is played with a display latency of two frame times. Each frame that arrives with an end-to-end delay greater than two frame times is discarded.

The top graph in Figure 6-1b shows the display queue length at each display initiation time. The bottom graph shows the display latency of the frame being displayed at each display initiation time. In addition, each latency bar is labeled with the acquisition time of the frame that is displayed at that display initiation time. In this example, frames *b*, *d*, and *f* arrive with end-to-end delays longer than two frame times and are discarded. Thus, use of the I-policy results in three gaps occurring in the playout at display initiation times 4,6, and 8.

Figure 6-1c shows the effect of executing the E-policy. Where the I-policy held frames in the queue until they could be played with a particular display latency, the E-policy plays a new frame at each display initiation time as long as the display queue is not empty. Furthermore, frames are never discarded; each frame that arrives is put into the display queue. Thus, frame *a* is played at the first display initiation time after it arrives (*i.e.*, display initiation time 3). A gap occurs at display initiation time 4 because frame *b* has not yet arrived. When frame *b* does arrive, it is placed in the display queue and is eventually played at display initiation time 5. As a result, it is played with a display latency of 3 frame times. Furthermore, each succeeding frame is also played with a display latency of

3 frame times. As long as frames continue to arrive with an end-to-end delay less than 3 frame times, there will be no gaps.

The example shown in Figure 6-1 illustrates an advantage of the E-policy. The E-policy starts playing frames with the lowest possible initial display latency and then adjusts display latency upward in response to delay jitter. The overall effect of the E-policy is to find a display latency that is sufficient to play frames without gaps by dynamically adjusting the latency to be higher than any end-to-end delay yet observed.

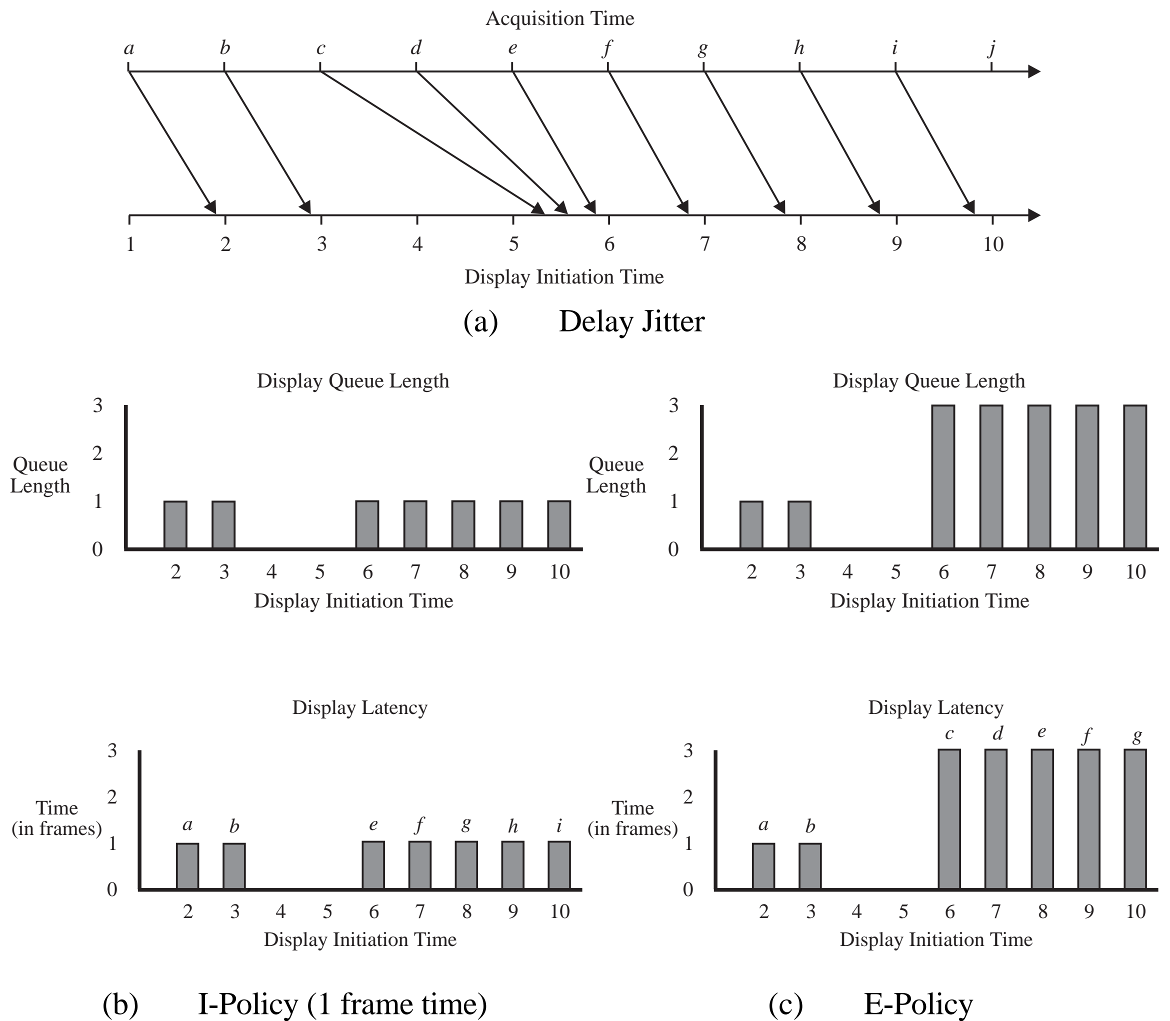

Figure 6-2: I-Policy and E-Policy with Occasional Delay Jitter

Figure 6-2 illustrates a situation in which the I-policy performs better than the E-policy. In this example, the display latency parameter of the I-policy is one frame time. All frames except frames *c* and *d* arrive with an end-to-end delay of less than one frame time and

experience negligible delay jitter. Frames *c* and *d* arrive late because of some temporary increase in network activity. Each policy results in gaps at display initiation times 4 and 5. However, the I-policy plays frames after the gap with low latency while the E-policy plays frames after the gap with higher latency. Under the E-policy, a single "burst" of activity on the network that causes a few frames to arrive late results in a permanent increase in display latency.

Overall then, the effect of both the I-policy and the E-policy is to choose a display latency at which to play frames, either explicitly in the case of the I-policy, or implicitly in the case of the E-policy. A good choice for display latency will depend on many factors. First, the acceptable rate of gaps and the acceptable display latency may vary depending on the application (*e.g.*, the transmission of speech may have different gap and latency requirements than the transmission of music) and the current requirements of the user (*e.g.*, a surgeon viewing an operation will have different requirements than viewers of a televised lecture). Second, the display latency required to maintain an acceptable gap-rate will depend on the expected level of delay jitter, which will vary as a result of congestion in the network. The dynamic nature of these factors motivates the design of a delay jitter management policy that dynamically changes display latency to adapt to new requirements and conditions.

6.3 Queue Monitoring

Consider an oracle that has perfect knowledge of the end-to-end delays of future frames and hence can choose the best display latency at which to play each frame. Such an oracle can adjust display latency in response to changes in delay jitter (perhaps due to changes in network congestion) in order to achieve the best possible balance between display latency and gaps. Display latency can be adjusted upward by artificially introducing a gap (*i.e.*, delaying the playout of the next frame) and can be adjusted downward by discarding frames.

If it is assumed that the delay jitter in the near future can be predicted by observing the delay jitter in the recent past, it is possible to construct delay jitter management policies that are approximations of the oracle. (This is analogous to the working set concept of page replacement in virtual memory management.) The Naylor and Kleinrock policy for choosing a display latency that is described in Chapter 1 is one example of such a policy.

This policy, however, is difficult to implement because accurately measuring end-to-end delays at runtime requires synchronized clocks.

Instead of measuring end-to-end delays, it is possible to directly measure the impact of delay jitter at a receiver by observing the length of the display queue over time. Once every frame time, a frame is removed from the display queue to be played (*e.g.*, for video frames displayed at 30 frames per second, a frame is removed every 33 ms.). Since frames are also acquired and transmitted once per frame time, *on average* one frame will arrive and be placed in the display queue and one frame will be removed from the display queue during each frame time. If end-to-end delays are constant, the queue length measured at each display initiation time should be constant (as it was between display initiation times 6 and 10 in the example shown in Figure 6-2c). If delay jitter results in a frame arriving with a longer end-to-end delay, then it is possible that no new frames will arrive between successive display initiation times. In that case, the length of the display queue will decrease by one frame (*e.g.*, display initiation time 6 in Figure 6-1c). If delay jitter results in a frame arriving with a shorter end-to-end delay, then more than one frame may arrive between successive display initiation times, and the length of the display queue will increase (*e.g.*, display initiation time 6 in Figure 6-2c).

Over time, the length of the display queue will vary depending on the range of end-to-end delays encountered by frames. If the level of delay jitter in the near future will be the same as the level in the recent past, then while end-to-end delays may vary, they will not vary outside the range that has been observed recently. This implies that in the near future, the length of the display queue will remain at least as long as the minimum length that has been observed in the recent past.

The assumption that delay jitter in the near future will be about the same as that in the recent past can be used to determine if a frame can be discarded from the display queue in order to reduce display latency without causing more gaps. As long as the display queue contains at least one frame at each display initiation time, there will be no more gaps in the playout. If the minimum display queue length observed recently was at least two frames, then it can be assumed that after discarding a frame, the minimum queue length observed in the near future will be at least one. Thus, a frame can be discarded without causing additional gaps.

A policy for decreasing display latency based on observing queue lengths has been used to govern the behavior of the audio display queue in the Pandora system [28]. Whenever

frames are added to the display queue (called the *clawback buffer*), the length of the queue is checked against a target value. In the Pandora system, the target is 2 frames (in Pandora, each audio frame corresponds to 2 ms. of audio data). If the length of the display queue is greater than this target for a sufficiently long interval (8 seconds), incoming audio frames are discarded. Because this has the effect of shortening the display queue, audio data that arrives after this time will be played with a lower display latency.

```
var
   threshold : array of integer;
   above     : array of integer;

for i := max_queue_length downto 2 do
   if length(display_queue) > i then
      above[i] := above[i] + 1
   else
      above[i] := 0
   end if;

   if above[i] >= threshold[i] then
      buffer := remove_queue(display_queue);
      free(buffer);
      for j := 2 to max_queue_length do
         above[i] := 0
      end for;
      exit loop
     end if
end for
```

Figure 6-3: Queue Monitoring Procedure

I propose a display queue management policy called *queue monitoring* that is a variation of the policy used in Pandora. In this policy, a *threshold value* is defined for each possible display queue length. The threshold value for queue length *n* specifies a duration (measured in frame times) after which, if the display queue has continuously contained more than *n* frames, it will be concluded that display latency can be reduced without increasing the frequency of gaps. Since I am making the natural assumption that large variations in end-to-end delay are expected to occur infrequently, and small variations are expected to occur much more frequently, threshold values for long queue lengths specify short durations while those for short queue lengths specify long durations (this assumption is validated by the histograms of delay jitter given in Figures 7-2, 7-4, and 7-6).

In the implementation of queue monitoring, an array of counters and threshold values is associated with the display queue. Each display initiation time, the queue monitoring procedure illustrated in Figure 6-3 is performed. First, when the display queue has length

m, counters 2 through m–1 are incremented and all other counters are reset. Then, if any counter exceeds its associated threshold value, all the counters are reset and the oldest frame is discarded from the display queue. Once the queue monitoring procedure has been performed, the oldest remaining frame is removed from the display queue and played.

An important principle in this implementation is that the thresholding operation will never discard frames unless the display queue contains more than two frames. The last frame in the display queue should never be discarded because there must be a frame available for display after the thresholding operation completes. Similarly, if the second-to-last frame in the display queue were discarded, then even minute delay jitter could potentially cause a gap. For example, this can occur in a situation where a frame arrives immediately before the thresholding operation and results in a display queue with length 2. If one of those frames is discarded and the other is displayed, then the queue would be empty. Then, if the next frame has a slightly larger end-to-end delay, it might not arrive in time to be displayed. Therefore, I only consider queue monitoring policies with thresholds defined for queue lengths greater than two.

6.4 Summary

In this chapter, I have described several policies for managing the tradeoff between the display of continuous media with low display latency and the display of continuous media with few gaps. I began by asserting that policies for managing this tradeoff could be implemented as policies for managing the display queue. I then described two queue management policies from the literature, the I-policy and the E-policy, and illustrated the effect of these policies with several examples. Finally, I defined a queue management policy called queue monitoring that was designed to combine the advantages of the I- and the E-policies. In the next chapter, I evaluate the relative performance of these three policies.

Chapter VII
Evaluation of Delay Jitter Management Policies

7.1 Introduction

This chapter presents an empirical study of the set of policies described in Chapter 6 for managing the display queue in the presence of delay jitter. In the study, the workstation-based video conferencing application described in Chapter 2 was run a number of times in several different network environments. During execution, traces of the end-to-end delays experienced by frames were recorded. These traces were then used as input to a simulator which determined the effect that applying each policy would have had on the quality of the conference.

The goal of the study was to gauge the effectiveness of the set of delay jitter management policies in campus-sized internetworks. In principle, the effectiveness of each policy should be independent of the particular type of continuous media data on which it operates. In practice however, differences in continuous media data types affect the behavior of the policies. The most important difference is frame rate; data displayed at a high frame rate is more sensitive to delay jitter than data displayed at a low frame rate. For example, under any policy, a 200 ms. variation in the end-to-end delays experienced by frames will have little effect when frames are displayed at a rate of one per second. However, when frames are displayed at a rate of 60 frames per second, then a 200 ms. variation in delay has a much greater effect; under the queue monitoring policy, such a variation in delay could cause the length of the display queue to decrease by 13 frames. In the workstation-based video conferencing application, audio is displayed at a higher frame rate than video; thus audio is more sensitive to delay jitter. As a result, I have chosen to study the effect of the I-policy, the E-policy, and the queue monitoring policy on the display of audio frames.

To evaluate the performance of the policies over a range of networks, I performed a test suite in three different network environments. The first was the internetwork consisting of 16Mb Token Rings and 10Mb Ethernets that serves as the main network supporting the

Computer Science department at the University of North Carolina. The other two network environments were located on the campus of IBM in Research Triangle Park, North Carolina. First, the test suite was performed on a 4 Mb Token Ring serving a single floor of a building. Next, the test suite was repeated using two 4 Mb floor rings connected by a 16 Mb Token Ring serving as the backbone network for all the buildings on the IBM-RTP campus.

In Section 7.2, I begin by describing the study in detail. In Section 7.3, I discuss the metrics I propose for evaluating the performance of the delay jitter management policies and the problems inherent in formulating such metrics. In Section 7.4, I compare the performance of queue monitoring to that of the I- and E- policies. In Section 7.5, I explore the effect of the threshold parameter of the queue monitoring policy.

7.2 Description of the Study

In this section, I present a detailed description of the data collected in the study, as well as a description of the network environments in which the study was conducted. In each network environment, I executed the workstation-based videoconferencing application several times; each complete execution is referred to as a *run*. During each run, the application acquired, transmitted, and displayed 60 frames per second audio and 30 frames per second video and recorded a trace of the acquisition time and the arrival time of each audio frame as well as the time of each VBI logical interrupt at the display (*i.e.*, the times at which new audio frames were displayed). Audio frames were transmitted in individual packets and video frames were broken into fragments which would fit into a single packet on an ethernet (*i.e.*, 1350 bytes of video data per fragment).

After each run, the resulting trace was adjusted to account for the lack of clock synchronization. The adjustment was based on two values. First, before each run I executed a simple protocol to measure the difference between the time at the acquisition machine and the time at the display machine. Second, I measured the ratio between the clock rates at the acquisition and display machines in a separate experiment. For each pair of machines, this ratio was found to be nearly fixed (*e.g.*, the ratio of the clock rates for the pair of machines used in the UNC experiments was 0.999987). Thus, the frame generation times, which were measured using the clock on the acquisition-side, were converted to display-side times by multiplying by the ratio of the clock rates and adding the initial difference in clock times.

Each trace was also adjusted to account for packets lost in the network. Any audio frame that never arrived was assumed to have been generated 1/60th of a second after the preceding audio frame and to have arrived at the display at the same time as the next audio frame that arrived (*i.e.*, if frame N was lost, then it was assumed to have been generated one frame time after frame N-1 and to have arrived at the display at the same time as frame N+1). This adjustment was used in order to examine the delay jitter encountered in real-world networks in isolation from the loss encountered in real-world networks; the choice that a lost frame is assumed to arrive with the next frame reflects an assumption that a forward error correction scheme would be used to correct errors.

Finally, the adjusted traces were used as input to a trace-driven simulation of the display-side of a conference. For a given display queue management policy and the sequence of arrivals and display initiation times in the trace, the simulator determined which frame would have been displayed at which display initiation time. The output of the simulator was the average display latency and average gap rate that would have resulted from applying the policy during the run.

7.2.1 Description of the UNC Network Environment

The first set of runs was performed using the main network supporting the Computer Science department at the University of North Carolina. This network consists of several 10 Mb Ethernets and 16 Mb token rings interconnected by bridges and routers. It supports approximately 400 UNIX workstations and Macintosh personal computers. The workstations share a common filesystem using a mix of NFS and AFS and run an overall application mix that should be typical of most academic computer science departments. In this set of runs, each packet was routed across a lightly-loaded token ring to a gateway, through a segment of the departmental ethernet to a bridge, through a second segment of the departmental ethernet to another gateway, and back across the same token ring to the display machine.

Twenty-four runs, each lasting 10 minutes, were performed over the course of a typical day (between 6am and 5pm) covering lightly and heavily loaded periods. Four additional runs were performed during nightly backups (between midnight and 1am). Figure 7-1 gives some basic data on the variability in end-to-end delays encountered by audio frames during the 28 runs. "Time of Day" is the time the run was initiated. Average and maximum delays are calculated from the end-to-end delays experienced by audio frames (recall that end-to-end delay is defined as the elapsed time between acquisition of the

frame and its arrival at the display queue). Lost and duplicate frames are counts of lost and duplicated packets which contained an audio frame. No out of order packets were observed.

Run	Time of Day	Avg. Delay ms.	Max. Delay ms.	Lost Frames	Duplicate Frames
1	06:03	38	76	1	0
2	06:25	38	88	3	0
3	06:36	37	171	5	0
4	06:47	37	105	1	0
5	08:03	38	115	1	0
6	08:14	37	73	2	0
7	08:25	38	184	7	0
8	08:36	39	157	1	0
9	10:02	41	186	23	0
10	10:16	40	124	4	0
11	10:31	41	213	7	0
12	10:49	40	140	6	0
13	11:57	39	110	5	0
14	12:08	41	138	5	0
15	12:19	41	133	3	0
16	12:34	40	187	11	0
17	14:02	41	189	11	0
18	14:13	42	141	3	0
19	14:42	39	107	4	0
20	14:54	40	131	12	0
21	16:01	39	171	9	0
22	16:21	39	128	2	0
23	16:33	39	86	2	1
24	16:55	42	242	14	1
25	00:05	38	80	4	0
26	00:16	38	128	0	0
27	00:27	38	134	8	0
28	00:38	38	83	2	0

Figure 7-1: Basic Data (UNC Network)

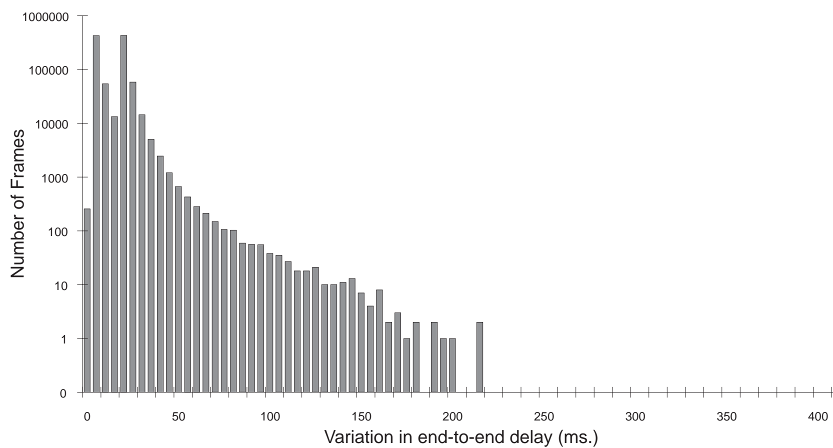

Figure 7-2: Distribution of End-to-End Delay Jitter (UNC Network)

Figure 7-2 provides a more detailed look at the 28 runs. This figure illustrates the distribution of end-to-end delay jitter experienced by audio frames. For each run, the delay jitter of an audio frame is defined by subtracting the minimum end-to-end delay observed during the run from the end-to-end delay of the frame. The y-axis shows a count plotted on a log scale of the number of frames with delay jitter within each 5 ms. interval (*e.g.*, a count of frames with end-to-end delay jitter of 0 ms. - 5 ms., 5 ms. - 10 ms., 10 ms. -15 ms., etc.).

7.2.2 Description of the IBM-RTP Floor Network

The next set of runs was performed using the network that supports a floor of an office building on the IBM campus in Research Triangle Park, North Carolina. This network is a single 4 Mb Token Ring, connected to the rest of the campus network by a bridge. It supports approximately 50 PS/2 workstations. In this set of runs, each packet was simply sent from the acquisition machine to the display machine across this one ring. Fifteen runs, each lasting 5 minutes, were performed on this network. These runs were performed between 9am and 5pm over several days. Figure 7-3 gives some basic data for each run and Figure 7-4 illustrates the distribution of end-to-end delay jitter experienced by audio frames during each run.

7.2.3 Description of the IBM-RTP Campus Network

The third set of runs was performed using the campus internetwork at IBM-RTP. This network consists of a 16 Mb Token Ring which serves as the backbone and is connected by bridges to 4 Mb Token Rings supporting single floors of each campus building. In this set of runs, each packet was routed across a floor ring to a bridge, through the backbone to another bridge, and through a second floor ring to the display machine. Nineteen runs, each lasting 5 minutes, were performed on this network. These runs were performed between 9am and 5pm over several days. Figure 7-5 gives some basic data on each run and Figure 7-6 illustrates the distribution of end-to-end delay jitter experienced by audio frames during each run.

7.2.4 Summary of the Three Network Environments

The three network environments used here exhibit somewhat different characteristics. On the UNC departmental network, most frames arrive with very little delay jitter, but variation in delay of as much as 220 ms. was encountered. On the IBM floor network,

more frames experienced delay jitter in the 30 to 60 ms. range, but the largest variation encountered was only in the range of 110 ms. Finally, on the IBM campus network, the largest variation in delay encountered was in the range of 410 ms. Furthermore, while little data was lost in the UNC departmental network, and none was lost in the IBM floor network, data loss in the IBM campus network was significant.

Run	Time of Day	Avg. Delay ms.	Max. Delay ms.	Lost Frames	Duplicate Frames
1	11:57	38	116	0	0
2	14:23	36	110	0	0
3	15:23	38	101	0	0
4	16:04	41	99	0	0
5	09:48	40	93	0	0
6	15:05	43	104	0	0
7	17:16	52	100	0	0
8	11:22	44	101	0	0
9	10:22	45	111	0	0
10	09:58	48	135	0	0
11	15:46	46	112	0	0
12	10:07	38	92	0	0
13	12:49	50	103	0	0
14	13:23	58	120	0	0
15	14:46	56	105	0	0

Figure 7-3: Basic Data (IBM-RTP Floor)

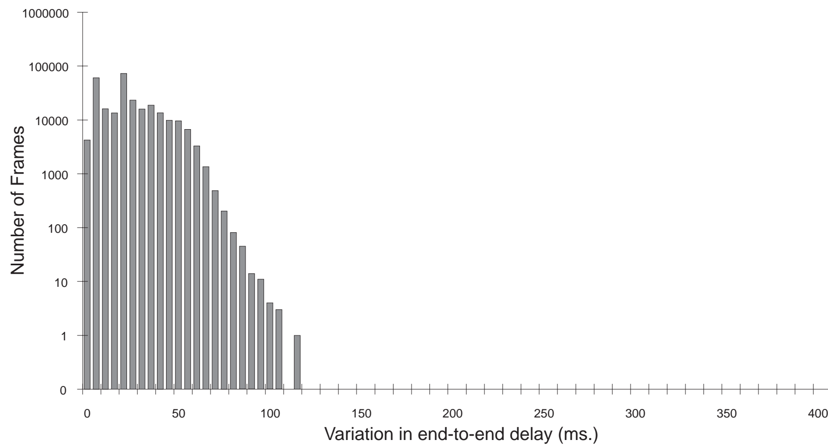

Figure 7-4: Distribution of End-to-End Delay Jitter (IBM-RTP Floor)

Run	Time of Day	Avg. Delay ms.	Max. Delay ms.	Lost Frames	Duplicate Frames
1	13:31	39	367	113	0
2	14:15	46	179	0	0
3	15:22	44	368	107	0
4	13:47	42	87	0	0
5	14:26	48	334	105	0
6	15:25	46	175	8	0
7	10:08	69	429	765	0
8	10:50	45	131	0	0
9	12:31	44	181	12	0
10	14:42	62	367	204	0
11	16:07	71	364	333	0
12	10:23	45	160	3	0
13	10:58	69	255	2	0
14	12:47	42	161	0	0
15	13:49	51	314	94	0
16	14:44	38	200	21	0
17	15:41	48	245	19	0
18	10:12	67	350	43	0
19	12:25	42	130	0	0

Figure 7-5: Basic Data (IBM-RTP Campus)

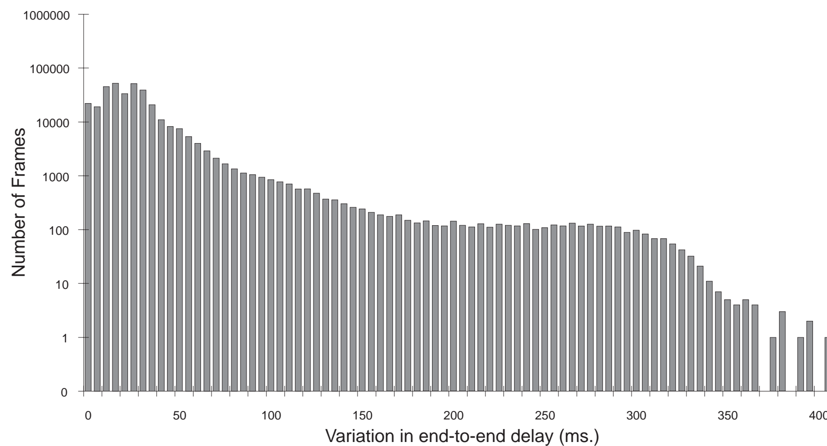

Figure 7-6: Distribution of End-to-End Delay Jitter (IBM-RTP Campus)

7.3 Evaluating Delay Jitter Management Policies

In the remainder of this chapter, I evaluate the effectiveness of the I-, E-, and queue monitoring policies at managing the effect of delay jitter on the quality of audio in a workstation-based videoconference. For each policy and each run, I use simulation based on the traces described in Section 7.2 to determine which frames would have been discarded and the display latency at which each remaining frame would have been played assuming the policy had been applied during that run. From this, I determine the average

display latency and gap rate for each policy on each run. Before these results can be used to compare the relative performance of the three policies, two issues must be addressed: what is the precise definition of a gap, and what metric should be used to determine if one policy has performed better than another?

7.3.5 Gaps

In Chapter 1, a gap was defined as the event that occurs when an application is unable to play the next frame when the display of the preceding frame is complete. Such an event can occur for several reasons. In [11], Gruber and Strawczynski divide gaps in the playout of audio into two types: open and closed. An *open gap* occurs when frame N is played, followed by a frame time of silence, followed by frame $N+2$. Thus, the gaps encountered when using the I-policy are open gaps. A *closed gap* occurs when frame N is played immediately followed by frame $N+2$. Thus, the gaps encountered when the queue monitoring policy reduces latency by discarding frames are closed gaps. A third type of gap, which I will refer to as a delay gap, is encountered when using the E-policy and the queue monitoring policy. A *delay gap* occurs when frame N is played, followed by a frame time of silence, followed by frame $N+1$.

Through informal observation, I have concluded that both open gaps and delay gaps cause a reduction in the perceived quality of audio. For open gaps, a study by Gruber and Strawczynski confirms this conclusion[8]. In my study, both open gaps and delay gaps are counted as gaps.

In contrast, I do not count closed gaps. While Gruber and Strawczynski's study does not address the effect of closed gaps on quality at a frame rate of 60 frames per second, it does provide data on closed gaps at 15 frames per second; a gap rate of 3.6 gaps per minute does not result in a noticeable decrease in quality. From this result, and supported by informal observations, I have concluded that for the rate at which queue monitoring reduces latency in my experiments (at most 6 frames per minute in the QM-600 policy defined below), the closed gaps introduced by queue monitoring should be undetectable.

---

[8]Participants in the study were asked to rate the quality of audio playout on a scale of 1 to 5. For a frame rate of 60 frames a second, playout without gaps resulted in a quality rating of 4.1, while a gap rate of 14.1 gaps per minute resulted in a rating of 3.5, a decrease in perceived quality of 15%.

7.3.6 Comparison Rule

To evaluate the effectiveness of a delay jitter management policy, it would be useful to have a metric that determined the display quality of a conference performed using that policy. Clearly, if policy *A* results in lower display latency and less gaps than policy *B*, it is performing better. However, if policy *A* results in a lower display latency and a higher gap rate as compared with policy *B*, which has performed better?

In their study of the I-policy and the E-policy [37], Naylor and Kleinrock answer this question using a straightforward comparison rule. They propose a quality metric in which display quality is the normalized Euclidean distance from the origin in the *DG* plane, where *D* is the display latency, *G* is the gap rate, and *D* and *G* are normalized by two constants, *d* and *g*. These two normalization constants are intended to be threshold values of display latency and gap rate, above which quality degrades rapidly.

Unfortunately, this comparison rule does not address many of the factors that affect the quality of audio and video display. These factors include not only the display latency and the gap rate, but also the resolution of the display, the user's particular requirements for audio and video, the distribution of gaps throughout the measurement interval, the number of display latency changes, and the distribution of periods of high and low display latency throughout the interval. A better standard for comparing policies would take each of these factors into account.

More importantly, the Naylor and Kleinrock comparison rule is based on an assumption that there is a direct tradeoff between gap rate and display latency. Any conclusions about the relative effect of two policies derived using this comparison rule would be extremely sensitive to the validity of this assumption and to the particular choice of the normalization constants. Since these assumptions are not necessarily justified, conclusions drawn using this comparison rule are potentially misleading. Nevertheless, in order to provide some insight into the data, it is useful to adopt some standard of comparison. Therefore, I have adopted a simple, conservative and arbitrary comparison rule for the analysis in this chapter.

My comparison rule is based on two measurements: average display latency and average gap rate. I assume that differences in display latency of less than 16.5 ms. (*i.e.*, a single audio frame time) and differences in gap rate of less than one every 15 seconds (*i.e.*, 4 gaps per minute) are not significant. My comparison rule declares policy *A* to have done

better than policy *B* if it is better in one dimension and the same or better in the other dimension. Two policies are declared to have done equally well if they are the same in both dimensions and are declared to be incomparable if each has done better in one dimension.

Given this comparison rule, I can evaluate and compare the effectiveness of policies for a particular run. However, it is still difficult to compare results of multiple runs. One fundamental difficulty arises because the video hardware that acquires frames at the sender is not synchronized with the display at the receiver. To illustrate the effect this has on display latency, assume there is no end-to-end delay (*i.e.*, acquisition and arrival of frames are simultaneous). Despite this fact, an application must wait until the next display initiation time (*i.e.*, the next VBI logical interrupt) to display each new frame. Depending on the synchronization difference between the video hardware acquiring the frames and the display, each frame may have to wait up to one frame time before being displayed. This synchronization time is a random variable and varies between runs. Therefore, when comparing results of multiple runs, differences in latency of as much as a frame time are not significant.

The second difficulty in comparing multiple runs arises from my working definition of the I-policy. As described in Chapter 6, the I-policy should play frames at a constant display latency. However this would require that the clocks at the acquisition and display workstations be synchronized. In my work, I only assume synchronized clocks for measurement purposes (*i.e.*, I do not use synchronized clocks to guide the execution of the system). Therefore, I cannot implement the I-policy. Instead, I implement a variant of the I-policy which buffers the first frame for a fixed number of frame times before displaying it and then displays all subsequent frames with the same display latency. The effect of this definition is to make the display latency enforced by a particular I-policy during a run a function of the end-to-end delay of the first frame that is received (*i.e.*, a random variable).

The goal of the study presented in this chapter is to determine which of several policies results in the best quality playout over a range of network conditions. Because of the difficulties involved in comparing the results of multiple runs, and because my comparison rule determines relative, rather than absolute performance, I restrict direct comparisons to determining the relative performance of two policies on a single run. This allows me to conclude only that one policy outperforms another on a particular run. To show that one

policy outperforms another in general, I must show that it performs better on some runs and as well or better on all runs. This method of pairwise comparison is the basis of the performance evaluations presented in the remainder of the chapter.

7.4 Comparison of Queue Monitoring to the I- and E- Policies

In this section, I compare the performance of the queue monitoring policy with the performance of the I- and E-policies. The E-policy used here is exactly as it was described in Chapter 6. The I-policy and the queue monitoring policy used here require further elaboration.

As described in Chapter 6, the I-policy is parameterizable; it plays all frames at a specified display latency. However, as mentioned above, because the implementation does not rely on synchronized clocks, the application can only approximate the I-policy. Thus I use a variant of the I-policy in which the display latency parameter specifies a number of frame times for which the first frame is buffered after it arrives at the display; all subsequent frames are played with the same display latency.

For each network environment, I have arbitrarily chosen the parameter of the I-policy to reflect a desired average gap rate of less than 4 gaps per minute[9]. To set this parameter for a particular environment, I determined the value of delay jitter for which, over all the measurements of end-to-end delay jitter taken in that environment, less than 4 out of every 3600 frames arrived with greater delay jitter (note that this means that on some runs, significantly more than 4 gaps per minute will be encountered using this value). This value was 60 ms. for the UNC departmental network, 75 ms. for the IBM-RTP floor network, and 305 ms. for the IBM-RTP campus network. Thus to achieve an average gap rate of less than 4 gaps per minute in these network environments, the I-policy should have a parameter of 4, 5, and 19 frame times respectively (*i.e.*, $\lceil 60/16.5 \rceil = 4$, $\lceil 75/16.5 \rceil = 5$, $\lceil 305/16.5 \rceil = 19$). These are the values used for the analysis in this section.

---

[9]This is a conservative choice for a desired gap rate. Nevertheless, the resulting display latency is relatively small for both the UNC departmental network and the IBM-RTP floor network. However, this desired gap rate leads to a very large display latency for the IBM-RTP campus network. This shows that there is too much delay jitter in the IBM-RTP campus network to support display at a fixed latency with few gaps.

The queue monitoring policy is also parameterizable; threshold times must be specified for each queue length. In this section, I compare a simple queue monitoring policy to the I- and E- policies. Each queue length greater than two is assigned a threshold of 10 seconds (600 frame times). The effect of these threshold settings is to reduce display latency by one audio frame time (16.5 ms.) whenever the display queue contains more than 2 audio frames for 600 continuous frame times (10 seconds).

Figure 7-7 shows the simulation results for each of the 28 runs on the UNC departmental network. In the table, the I policy is labeled I-4, the E-policy is labeled E and the queue monitoring policy is labeled QM-600. For each policy, the table shows the resulting average display latency (in ms.) and the average gap rate (in gaps/minute). For each run, the rightmost columns show the comparison between the queue monitoring policy and the other policies (using the comparison rule defined in Section 7.3). A '+' indicates that queue monitoring did better, a '0' means the two were equivalent, a '-' means that queue monitoring did worse, and an 'x' means the two policies were incomparable. The total number of runs for which QM was better, equivalent, worse, or incomparable is summarized at the bottom of the table.

The results in Figure 7-7 show that, with respect to my comparison rule, the QM-600 policy performs as well or better than both the I-policy and the E-policy on every run. On several runs, the difference is striking. For instance, on run 3, queue monitoring resulted in a display latency 100 ms. less than that produced by the E-policy while producing the same gap rate. From this I surmise that run 3 is probably an example of the poor behavior of the E-policy that was abstractly illustrated in Figure 6-2. On run 24, queue monitoring resulted in a display latency comparable to that produced by the I-policy, but with a gap rate 4 times smaller. From this, I surmise that a portion of run 24 exhibited the poor behavior of the I-policy illustrated in Figure 6-1.

From the results in Figure 7-7, I conclude that over the range of network conditions observed in the UNC departmental network, the use of queue monitoring as the delay jitter management policy was more effective than either the I-policy or the E-policy. Figure 7-8 shows that queue monitoring also performed better than the I-policy and the E-policy over the range of network conditions observed on the IBM-RTP floor network.

Run	I-Policy (I-4) Latency ms.	I-Policy (I-4) Gaps /min.	E-Policy Latency ms.	E-Policy Gaps /min.	QM-600 Latency ms.	QM-600 Gaps /min.	QM vs. I-4	QM vs. E
1	114	0.0	80	0.2	73	0.2	+	0
2	108	0.0	80	0.3	70	0.4	+	0
3	102	1.5	178	0.9	76	0.9	+	+
4	99	0.1	104	0.5	69	0.5	+	+
5	104	0.1	97	0.5	73	0.5	+	+
6	103	0.0	83	0.3	70	0.3	+	0
7	107	0.8	134	0.9	83	1.4	+	+
8	96	1.5	106	0.8	83	0.9	0	+
9	114	5.8	192	0.9	111	3.4	0	+
10	104	1.0	130	0.6	90	1.3	0	+
11	99	3.4	150	1.1	102	2.9	0	+
12	104	1.3	137	0.7	87	1.6	+	+
13	101	0.4	102	0.5	85	1.0	0	+
14	105	1.1	110	0.6	94	1.5	0	0
15	109	0.7	120	0.6	89	1.4	+	+
16	101	4.5	145	1.0	104	2.5	0	+
17	110	6.6	177	0.9	109	3.1	0	+
18	110	1.6	139	0.6	103	2.2	0	+
19	99	0.1	92	0.5	85	0.7	0	0
20	106	0.8	129	0.6	91	1.3	0	+
21	104	3.0	177	0.9	89	1.7	0	+
22	112	0.1	103	0.5	81	0.7	+	+
23	108	0.0	87	0.3	74	0.4	+	0
24	110	7.7	132	1.2	102	1.9	+	+
25	98	0.0	81	0.3	77	0.4	+	0
26	98	0.3	122	0.6	79	0.9	+	+
27	104	1.5	125	0.6	84	2.4	+	+
28	109	0.0	88	0.3	74	0.3	+	0
				QM Better			16	20
				QM Equivalent			12	8
				QM Worse			0	0
				Incomparable			0	0

Figure 7-7: Comparison of I, E, and QM Policies (UNC Network)

Figure 7-9 shows the results for the IBM-RTP campus network. On this network, queue monitoring never performed worse, and usually performed as well or better than the I-policy. On several runs however, my comparison rule judged the queue monitoring policy to be incomparable with the I-policy. Thus, I cannot conclude that, with respect to my comparison rule, queue monitoring performed better than the I-policy over the range of observed network conditions. However, looking deeper at the incomparable runs, it is clear that on most, queue monitoring produced a much lower display latency and a somewhat higher gap rate. On four of the five incomparable runs, the difference in gap rate produced by queue monitoring and by the I-policy was less than 7.8 gaps per minute, while the difference in display latency was as much as 219 ms. On the fifth incomparable

run, queue monitoring produced a slightly higher display latency, but 89 fewer gaps per minute. Thus, even on the incomparable runs, queue monitoring resulted in reasonable behavior that, intuitively, is probably better than and almost certainly not worse than that produced by the I-policy.

With respect to the E-policy, queue monitoring performed as well or better in the results for the IBM-RTP campus network on all but run 1; on that run, queue monitoring resulted in a display latency 46 ms. less than that produced by the E-policy, but also resulted in 4.2 more gaps per minute. So again, even on the incomparable run, queue monitoring resulted in reasonable behavior that is probably better than, and almost certainly no worse than that produced by the E-policy.

Run	I-Policy (I-5)		E-Policy		QM-600		QM	QM
	Latency ms.	Gaps /min.	Latency ms.	Gaps /min.	Latency ms.	Gaps /min.	vs. I-5	vs. E
1	106	4.2	120	1.2	116	2.0	0	0
2	108	0.0	96	1.0	72	1.0	+	+
3	104	0.0	103	1.0	75	1.0	+	+
4	115	0.0	93	0.8	72	0.8	+	+
5	113	0.0	90	0.8	82	2.2	+	0
6	103	0.4	97	1.2	85	2.0	+	0
7	108	0.0	104	1.0	95	1.6	0	0
8	107	0.0	98	1.0	88	1.6	+	0
9	114	0.0	104	1.0	88	2.0	+	0
10	106	5.8	139	1.4	109	2.6	0	+
11	108	0.2	120	1.2	100	2.6	0	+
12	108	0.0	93	1.0	80	2.2	+	0
13	107	0.0	85	1.0	82	1.2	+	0
14	116	0.2	132	1.2	104	1.6	0	+
15	102	0.2	104	1.2	102	1.4	0	0
				QM Better			9	6
				QM Equivalent			6	9
				QM Worse			0	0
				Incomparable			0	0

Figure 7-8: Comparison of I, E, and QM Policies (IBM-RTP Floor)

Run	I-Policy (I-19) Latency ms.	I-Policy (I-19) Gaps /min.	E-Policy Latency ms.	E-Policy Gaps /min.	QM-600 Latency ms.	QM-600 Gaps /min.	QM vs. I-19	QM vs. E
1	343	4.8	283	5.0	237	9.2	x	x
2	350	0.0	136	1.8	114	3.4	+	+
3	339	7.4	188	4.6	162	5.6	+	+
4	352	0.0	94	0.8	89	1.0	+	0
5	340	2.0	157	4.4	144	5.0	+	0
6	346	0.0	231	2.6	128	3.4	+	+
7	336	95.6	447	5.2	359	6.4	x	+
8	343	0.0	126	1.4	105	3.0	+	+
9	347	0.0	152	2.2	128	4.2	x	+
10	337	2.4	349	4.2	302	7.6	x	+
11	346	6.8	430	5.0	352	9.0	0	+
12	349	0.0	144	1.6	120	2.8	+	+
13	344	0.0	234	2.8	180	4.0	+	+
14	344	0.0	144	1.6	89	2.4	+	+
15	339	0.4	264	4.2	209	8.2	x	+
16	327	0.0	127	2.6	108	4.0	+	+
17	340	0.0	203	2.8	147	4.0	+	+
18	338	2.6	174	4.8	148	5.8	+	+
19	348	0.0	112	1.4	100	2.4	+	0
				QM Better			13	15
				QM Equivalent			1	3
				QM Worse			0	0
				Incomparable			5	1

Figure 7-9: Comparison of I, E, and QM Policies (IBM-RTP Campus)

Overall then, I conclude that the queue monitoring policy performed better than either the I- or the E- policies over the wide range of network conditions observed in the three environments. In particular, queue monitoring always resulted in lower display latency than that produced by the E-policy, and only rarely resulted in display latencies higher than that produced by the I-policy. Quite often, queue monitoring resulted in *much* lower display latencies than either or both of the other policies. And yet, the gap rate produced by queue monitoring was less than 3.1 gaps per minute on the UNC departmental network, and less than 9.2 gaps per minute on the IBM-RTP campus network; presumably an acceptable gap rate. Thus, queue monitoring appears to have successfully adapted to the delay jitter encountered in a wide range of network conditions to produce low display latency and an acceptable gap rate[10].

---

[10]Note however that for the IBM-RTP campus network, these results depend heavily on the assumption that there was no data loss in the network; with the loss that was encountered in that network, the true gap rate would be much greater.

7.5 Effect of the Threshold Parameter

In this section, I investigate the effect of the threshold parameter on the effectiveness of the queue monitoring policy. In the previous section, I looked at one queue monitoring policy with a single threshold (*i.e.*, 10 seconds) defined for all queue lengths greater than two. The effect of using a single threshold is to reduce display latency by one audio frame time whenever the display queue contains more than two audio frames continuously for the specified number of frame times. In this section, I begin by looking at several queue monitoring policies which define a single threshold for all queue lengths.

7.5.7 Results for QM Policies With a Single Threshold

For each run in the three network environments, I simulated the queue monitoring policy with three thresholds: 120 frame times (1/2 second), 600 frame times (10 seconds), and 3600 frame times (60 seconds). Figures 7-10, 7-11, and 7-12 summarize the results. As would be expected, on each run the use of a range of threshold parameters resulted in a range of results; since it discarded frames fastest, a threshold of 120 frame times produced the lowest display latency and highest gap rate, while a threshold of 3600 frame times produced the highest display latency and the lowest gap rate. Thus, thresholds seem to be a useful tunable parameter for an application to select a balance between display latency and gaps that reflects its requirements.

Looking at the three queue monitoring policies in the context of each network environment, it is clear that a single threshold value is not necessarily optimal across all network environments. In general, QM-600 performed somewhat better than QM-3600, although in the IBM-RTP floor network, performance was equivalent on most runs. However, the performance of QM-600 relative to QM-120 varied over the network environments. On the UNC departmental network, QM-120 performed as well or better than QM-600 on every run. On the IBM-RTP floor network, QM-120 produced equivalent results to QM-600 on most runs, slightly better on one run, and was incomparable on one run. On the IBM-RTP campus network, QM-120 performed better on several runs, worse on one run, and was incomparable on most runs. However, on most of the incomparable runs in both IBM-RTP networks, QM-120 resulted in very high gap rates.

Run	QM-120 Latency ms.	QM-120 Gaps /min.	QM-600 Latency ms.	QM-600 Gaps /min.	QM-3600 Latency ms.	QM-3600 Gaps /min.	QM-600 vs. QM-120	QM-600 vs. QM-3600
1	66	0.2	73	0.2	80	0.2	0	0
2	69	0.4	70	0.4	74	0.3	0	0
3	69	0.9	76	0.9	115	0.9	0	+
4	66	0.5	69	0.5	83	0.5	0	0
5	71	0.5	73	0.5	83	0.5	0	0
6	70	0.3	70	0.3	81	0.3	0	0
7	74	1.4	83	1.4	119	1.1	0	+
8	75	1.2	83	0.9	97	0.9	0	0
9	90	5.8	111	3.4	161	1.1	-	+
10	79	3.6	90	1.3	110	0.6	0	+
11	84	4.4	102	2.9	140	1.1	-	+
12	77	2.3	87	1.6	113	0.8	0	+
13	73	1.7	85	1.0	96	0.7	0	0
14	80	3.4	94	1.5	104	0.7	0	0
15	79	2.6	89	1.4	106	0.9	0	+
16	82	5.9	104	2.5	130	1.2	-	+
17	89	6.7	109	3.1	148	1.4	-	+
18	86	4.7	103	2.2	126	0.8	-	+
19	74	1.6	85	0.7	91	0.5	0	0
20	77	2.7	91	1.3	105	0.8	0	0
21	76	2.7	89	1.7	130	1.0	0	+
22	79	1.0	81	0.7	92	0.6	0	0
23	74	0.4	74	0.4	81	0.3	0	0
24	88	4.2	102	1.9	128	1.2	0	+
25	66	0.4	77	0.4	81	0.3	0	0
26	69	1.5	79	0.9	94	0.6	0	0
27	74	3.0	84	2.4	108	0.9	0	+
28	74	0.3	74	0.3	75	0.3	0	0
					QM-600 Better		0	13
					QM-600 Equivalent		23	15
					QM-600 Worse		5	0
					Incomparable		0	0

Figure 7-10: QM Policies with Varying Thresholds (UNC Network)

Therefore, it appears that, while no one threshold setting performs best for all network environments, it is possible that an optimal threshold exists for each environment. Furthermore, QM-600 produces reasonable results in each environment. Thus, an overall delay jitter management policy could begin by using queue monitoring with a threshold setting that always produces reasonable behavior. Then over time the threshold setting could be adjusted to reflect long-term observations of network conditions. Furthermore, such a policy could be used to adapt to long-term changes in network conditions (*e.g.* changes due to network reconfiguration).

Run	QM-120		QM-600		QM-3600		QM-600	QM-600
	Latency ms.	Gaps /min.	Latency ms.	Gaps /min.	Latency ms.	Gaps /min.	vs. QM-120	vs. QM-3600
1	94	21.4	116	2.0	120	1.2	x	0
2	71	1.0	72	1.0	82	1.0	0	0
3	73	1.0	75	1.0	85	1.0	0	0
4	71	1.2	72	0.8	85	0.8	0	0
5	73	4.4	82	2.2	90	0.8	0	0
6	78	3.2	85	2.0	97	1.2	0	0
7	88	3.6	95	1.6	104	1.0	0	0
8	80	4.6	88	1.6	97	1.0	0	0
9	81	2.8	88	2.0	101	1.2	0	0
10	91	6.4	109	2.6	130	1.4	-	+
11	87	6.4	100	2.6	120	1.2	0	+
12	75	2.4	80	2.2	91	1.2	0	0
13	80	1.6	82	1.2	85	1.0	0	0
14	94	4.8	104	1.6	119	1.2	0	0
15	93	2.4	102	1.4	104	1.2	0	0
					QM-600 Better		0	2
					QM-600 Equivalent		13	13
					QM-600 Worse		1	0
					Incomparable		1	0

Figure 7-11: QM Policies with Varying Thresholds (IBM-RTP Floor)

Run	QM-120		QM-600		QM-3600		QM-600	QM-600
	Latency ms.	Gaps /min.	Latency ms.	Gaps /min.	Latency ms.	Gaps /min.	vs. QM-120	vs. QM-3600
1	136	14.8	237	9.2	280	5.4	x	+
2	95	5.8	114	3.4	134	2.0	-	+
3	112	7.8	162	5.6	186	4.8	-	+
4	85	1.0	89	1.0	94	0.8	0	0
5	118	8.0	144	5.0	154	4.6	-	0
6	93	8.0	128	3.4	207	2.6	x	+
7	256	18.2	359	6.4	427	5.2	x	+
8	87	6.8	105	3.0	123	1.4	-	+
9	105	11.6	128	4.2	148	2.4	x	+
10	225	26.2	302	7.6	342	4.4	x	+
11	285	28.6	352	9.0	410	5.2	x	+
12	102	10.8	120	2.8	140	1.8	x	+
13	143	14.2	180	4.0	228	2.8	x	+
14	80	2.8	89	2.4	126	1.8	0	+
15	133	17.0	209	8.2	253	4.8	x	+
16	88	6.2	108	4.0	126	2.8	-	+
17	116	11.6	147	4.0	196	2.8	x	+
18	133	7.8	148	5.8	165	5.2	0	+
19	88	6.8	100	2.4	112	1.4	+	0
					QM-600 Better		1	16
					QM-600 Equivalent		3	3
					QM-600 Worse		5	0
					Incomparable		10	0

Figure 7-12: QM Policies with Varying Thresholds (IBM-RTP Campus)

7.5.8 Results for QM Policies With Varying Thresholds

In the queue monitoring policies investigated so far, display latency was decreased if the length of the display queue was continuously greater than two for a specified time; otherwise the behavior of these policies was not dependent on the queue length. However, the general queue monitoring policy described in Section 6.3 was designed to reduce latency quickly when the display queue was long. In Section 7.2 it was shown that in the network environments used in this study frames do incur significant delay jitter; thus long display queues can be encountered.

For example, consider the histogram of delay jitter given in Figure 7-6; some frames arrive with an end-to-end delay 410 ms. greater than the minimum end-to-end delay encountered during the same run. If the queue monitoring policy (or the E-policy) were used during a run with that level of delay jitter, then at some point during a run the length of the display queue would be at least 25 frames. More interesting is the observation that in each of the histograms of delay jitter (Figures 7-2, 7-4, and 7-6) the number of frames incurring a particular level of delay jitter decreases rapidly as delay jitter increases (up to approximately 150 ms.). This observation motivates the use of a queue monitoring policy in which decreasing thresholds are defined for increasing queue lengths.

Thus, in this section I examine the performance of the general queue monitoring policy in which individual thresholds are defined for each queue length. These thresholds can be arbitrary, but for purposes of this study, I have defined a particular rule for setting the threshold values. This rule has two parameters: a threshold value for a queue of length 3 measured in frame times, referred to as the *base threshold*, and a decay factor which specifies a rate at which the thresholds decrease with increasing queue length. For example, a queue monitoring policy with a base threshold of 3600 and a decay factor of 2 would have the threshold values: 3600 for queues of length 3, 1800 for queues of length 4, 900 for queues of length 5, etc. (*i.e.*, the threshold for length 5 means that a display latency is decreased if the display queue contains 5 frames for at least 15 seconds). Figures 7-13, 7-14 and 7-15 summarize the results.

Run	QM-3600		QM-3600,2		QM-3600,3		QM-3600 vs. QM-3600,2	QM-3600 vs. QM-3600,3
	Latency ms.	Gaps /min.	Latency ms.	Gaps /min.	Latency ms.	Gaps /min.		
1	80	0.2	80	0.2	80	0.2	0	0
2	74	0.3	74	0.3	74	0.3	0	0
3	115	0.9	80	0.9	75	0.9	-	-
4	83	0.5	82	0.5	81	0.5	0	0
5	83	0.5	76	0.5	75	0.5	0	0
6	81	0.3	81	0.3	81	0.3	0	0
7	119	1.1	93	1.2	89	1.2	-	-
8	97	0.9	89	0.9	86	0.9	0	0
9	161	1.1	127	1.8	118	2.1	-	-
10	110	0.6	108	0.6	97	0.8	0	0
11	140	1.1	114	1.7	106	1.8	-	-
12	113	0.8	101	0.9	95	1.0	0	-
13	96	0.7	96	0.7	96	0.7	0	0
14	104	0.7	104	0.7	101	0.9	0	0
15	106	0.9	101	1.0	97	1.0	0	0
16	130	1.2	114	1.8	105	2.2	0	-
17	148	1.4	122	2.2	111	3.2	-	-
18	126	0.8	122	0.9	117	1.2	0	0
19	91	0.5	88	0.5	88	0.5	0	0
20	105	0.8	100	0.8	99	0.9	0	0
21	130	1.0	102	1.1	98	1.1	-	-
22	92	0.6	88	0.7	87	0.7	0	0
23	81	0.3	81	0.3	81	0.3	0	0
24	128	1.2	110	1.2	102	2.0	-	-
25	81	0.3	81	0.3	81	0.3	0	0
26	94	0.6	89	0.6	89	0.6	0	0
27	108	0.9	98	1.2	94	1.2	0	0
28	75	0.3	75	0.3	75	0.3	0	0
					QM-3600 Better		0	0
					QM-3600 Equivalent		21	19
					QM-3600 Worse		7	9
					Incomparable		0	0

Figure 7-13: QM Policies with Multiple Thresholds (UNC Network)

Run	QM-3600 Latency ms.	QM-3600 Gaps /min.	QM-3600,2 Latency ms.	QM-3600,2 Gaps /min.	QM-3600,3 Latency ms.	QM-3600,3 Gaps /min.	QM-3600 vs. QM-3600,2	QM-3600 vs. QM-3600,3
1	120	1.2	119	1.4	118	1.6	0	0
2	82	1.0	77	1.0	76	1.0	0	0
3	85	1.0	85	1.0	84	1.0	0	0
4	85	0.8	84	0.8	83	0.8	0	0
5	90	0.8	90	0.8	90	0.8	0	0
6	97	1.2	97	1.2	94	1.2	0	0
7	104	1.0	104	1.0	104	1.0	0	0
8	97	1.0	95	1.0	95	1.0	0	0
9	101	1.2	98	1.4	96	1.6	0	0
10	130	1.4	127	1.6	120	1.6	0	0
11	120	1.2	117	1.2	107	1.2	0	0
12	91	1.2	89	1.4	89	1.4	0	0
13	85	1.0	85	1.0	85	1.0	0	0
14	119	1.2	119	1.2	119	1.2	0	0
15	104	1.2	104	1.2	104	1.2	0	0
				QM-3600 Better			0	0
				QM-3600 Equivalent			15	15
				QM-3600 Worse			0	0
				Incomparable			0	0

Figure 7-14: QM Policies with Multiple Thresholds (IBM-RTP Floor)

For each run in the three network environments, I simulated the queue monitoring policy with a base threshold of 3600 frame times and with decay factors of one, two and three. Again, on each run using a range of parameters resulted in a range of results; a decay factor of 1 produced the highest display latency and lowest gap rate, and a decay factor of 3 produced the lowest display latency and the highest gap rate. Thus, the use of smaller thresholds for longer queue lengths seems to be a useful tunable parameter for an application to select a balance between display latency and gaps that reflects its requirements.

Again, looking at the three queue monitoring policies in the context of each network environment, it is clear that a single decay factor is not necessarily optimal across all network environments. In general, QM-(3600,2) and QM-(3600,3) performed somewhat better than QM-3600, although in the IBM-RTP floor network, performance was equivalent on every run. However, on the IBM-RTP campus network, QM-(3600,3) resulted in high gap rates. Thus it appears that no decay factor is optimal for all network environments. However, as is the case with the base threshold, it appears that an overall delay jitter management policy could adjust the decay factor to reflect long-term observations of network conditions.

Run	QM-3600		QM-3600,2		QM-3600,3		QM-3600 vs. QM-3600,2	QM-3600 vs. QM-3600,3
	Latency ms.	Gaps /min.	Latency ms.	Gaps /min.	Latency ms.	Gaps /min.		
1	280	5.4	229	9.2	162	11.6	-	x
2	134	2.0	123	2.6	117	2.6	0	-
3	186	4.8	165	5.4	134	6.2	-	-
4	94	0.8	94	0.8	94	0.8	0	0
5	154	4.6	148	4.8	135	6.2	0	-
6	207	2.6	137	2.8	122	3.2	-	-
7	427	5.2	353	6.8	272	14.4	-	x
8	123	1.4	119	1.4	115	1.8	0	0
9	148	2.4	139	3.0	122	5.6	0	-
10	342	4.4	299	8.6	236	20.8	x	x
11	410	5.2	348	8.6	294	23.6	-	x
12	140	1.8	127	2.2	120	3.6	0	-
13	228	2.8	190	3.2	166	4.8	-	-
14	126	1.8	106	2.0	101	2.0	-	-
15	253	4.8	199	9.2	158	12.4	x	x
16	126	2.8	119	3.2	110	3.6	0	0
17	196	2.8	162	3.4	140	5.6	-	-
18	165	5.2	161	5.4	153	5.6	0	0
19	112	1.4	112	1.4	110	1.4	0	0
				QM-3600 Better			0	0
				QM-3600 Equivalent			9	5
				QM-3600 Worse			8	9
				Incomparable			2	5

Figure 7-15: QM Policies with Multiple Thresholds (IBM-RTP Campus)

7.6 Discussion and Summary

In this chapter, I have presented the results of an empirical study of the delay jitter management policies presented in Chapter 6. Overall, the study showed that queue monitoring performed better than either the I-policy or the E-policy over the range of observed network conditions. Furthermore, the study showed that the queue monitoring policy was flexible and tunable; a range of threshold parameters produced a range of results.

While the study of queue monitoring and the other policies was performed only for audio, these policies apply equally to video and other types of continuous media. In addition, while these policies have been presented as operating on the display queue, they are not restricted to the display queue. In particular, they can be applied to any queue from which frames are removed periodically. Thus in the workstation-based videoconferencing application, queue monitoring is applied to the queue of video frames which have arrived at the display workstation and are waiting to be decompressed.

However, there are two reasons why it should not be assumed that good settings for the parameters of the queue monitoring policy for video can be based on good settings for audio. First, because the size of audio and video frames differs, they will experience different levels of delay jitter. Second, because the frame rate at which video frames are displayed differs from the frame rate at which audio frames are displayed, it requires greater delay jitter to cause a change in queue length.

Chapter VIII
Conclusions and Contributions

8.1 Thesis Summary

Distributed applications that acquire and display live continuous media data (*e.g.*, audio and video) are subject to several timing constraints: operations on continuous media frames must often be executed within a narrow window of time, and the elapsed time between acquisition and display of frames must be reasonably short. Delay jitter (*i.e.*, variation in the time required to acquire, process, and transmit frames) causes difficulties in adhering to these constraints. There are two complementary approaches to addressing these difficulties. First, an application may reduce or eliminate delay jitter by carefully managing the process of acquiring, processing, transmitting, and displaying frames; however, this may require services from the operating system and network transport system that are not usually provided in general-purpose computing environments. Second, an application may adapt to the remaining delay jitter by playing frames at a sufficiently high display latency; however, high display latency may detract from the quality of the resulting playout.

The thesis of this dissertation is that a combination of these approaches is an effective solution to the problem of displaying continuous media in the presence of delay jitter. In the dissertation, I first demonstrated that it is possible to reduce delay jitter by designing, analyzing, and implementing the software at workstations that acquire or display continuous media as a real-time system with strict performance requirements. I then proposed and evaluated a policy called queue monitoring that dynamically adjusts display latency to accommodate the remaining delay jitter. In this dissertation, I have evaluated this combined approach using a workstation-based videoconferencing application that acquires audio and video at one workstation, transfers it over a network, and displays it at a second workstation.

The first part of the dissertation addressed the reduction of delay jitter through the use of hard-real-time design, analysis, and implementation techniques. Specifically, it was shown

that on the acquisition-side of the workstation-based videoconferencing application, each video frame is acquired, digitized, compressed, and delivered to the network transport system in bounded time. This was shown in four steps. First, I described an operating system kernel for the IBM PS/2 called YARTOS; the application executes on top of this kernel. Next, I defined an abstract model of real-time systems that was implementable using the programming model of YARTOS; for this abstract model, I developed a feasibility test to determine if the tasks that comprise a real-time system always execute prior to application-defined deadlines and within application defined mutual exclusion constraints. In the third step, I developed techniques for representing the application in terms of the abstract model; this allowed me to use the feasibility test to show that the deadline and mutual exclusion properties hold. Finally, I developed an axiomatic specification of that portion of the acquisition-side of the application that is responsible for acquiring, digitizing, and compressing video frames; from this specification, I derived the fact that each video frame is delivered to the network in bounded time. In addition, I argued that this analysis could be extended to show bounded-delay properties for audio frames on the acquisition-side and audio and video frames on the display-side.

The second part of the dissertation addressed the problem of accommodating delay jitter through the use of policies that manage the display queue (*i.e.*, the queue of frames waiting to be displayed). Three policies were considered, two from the literature and a new policy called queue monitoring. The queue monitoring policy operates by observing the length of the display queue over time; changes in queue length are a measure of delay jitter that is used to choose the display latency at which each frame is played. The performance of these policies was compared in an empirical study that used the workstation-based videoconferencing application to record the end-to-end delays experienced by audio frames transmitted via IP protocols over ethernets and token rings. The resulting traces were used as input to a simulator that determined the effect that applying each policy would have had on the quality of the audio playout. Overall, it was shown that queue monitoring could successfully adapt to the delay jitter incurred by audio frames in a wide range of network conditions. Furthermore, it was shown that the parameters of the queue monitoring policy provide a flexible method of tuning the performance of the policy to account for long-term changes in network conditions.

8.2 Conclusions

From this research, I conclude that techniques developed for designing, analyzing, and implementing hard-real-time systems can be successfully applied to applications that support continuous media. This allows the designers of distributed applications that support continuous media to assume that the only unbounded source of delay jitter is transmission over the network. Furthermore, I conclude that queue monitoring is an effective policy for ameliorating the effect of the delay jitter encountered in campus-sized networks on the display of continuous media. In particular, over the range of network conditions encountered in my study, the use of queue monitoring resulted in the lowest latency and fewest gaps of any of the policies studied.

8.3 Contributions

This dissertation makes contributions in several areas. First, I have expanded the toolkit of analysis techniques available to the designers of hard-real-time systems by developing a new formal model of real-time systems that addresses limitations found in traditional formal models. In previous models it was difficult to represent the behavior of hardware and software designed to be used in general-purpose environments. In particular, my model can represent interrupts, interrupt controllers, interrupt handlers, and synchronization primitives, as well as the sporadic (and/or periodic) tasks traditionally used to model real-time systems. Furthermore, my model allows designers to assign arbitrary deadlines to tasks. These properties are necessary if a formal model of real-time systems is to be useful in the design, analysis, and implementation of applications that support continuous media, which must often use hardware and software that was designed to be used in general-purpose environments.

Second, this dissertation provides a case-study of the design, analysis, and implementation of a significant real-time system. The design and implementation of the application show that it is possible to create significant real-time systems whose correctness can be shown through analysis. The separation of concerns enforced by the division of the analysis into the analysis of timing behavior and the analysis of logical correctness decouples reasoning about the architecture of a real-time system from assumptions about low-level details about how long tasks and actions require to execute, assumptions about scheduling, and assumptions about the existence of other tasks in the application. This property of the analysis shows that a similar analysis could be performed in a practical setting in which

those low-level details are subject to change over the course of the development of the system. Thus, the case-study presented here can contribute to the wider acceptance of formal techniques for designing and analyzing hard-real-time systems.

Third, the dissertation introduces queue monitoring, a policy for ameliorating the effect of delay jitter on the display of continuous media frames. Queue monitoring is flexible and general policy that can be applied in applications that support a variety of continuous media data types in the presence of delay jitter.

Fourth, the dissertation provides real-world data on the delay jitter that is experienced by continuous media data in campus-sized networks. In particular, since the software that acquired, transmitted, and received the data was implemented as a real-time system, the data on delay jitter was recorded without interference from arbitrary behavior of network and operating system software.

Finally, the dissertation provides a case study of the design of a continuous media application in an environment consisting of today's personal workstations, today's commercially available audio/video hardware, and today's networks (*e.g.*, ethernets, token rings, etc.). This design relies on few assumptions about the speed of processing or transmitting frames, or about the ordering of events. Thus, it can be applied to a variety of continuous media data types in a variety of environments.

In particular, this research will remain relevant in environments with faster machines, faster video compression technologies, and higher-speed data networks. Although faster hardware may be sufficient to support a single stream of video data in today's applications, tomorrow's applications will include more streams per application (*e.g.,* hundreds of participants in a video teleconference), much higher resolution pictures (*e.g.,* HDTV), and faster frame rates (*e.g.,* 60 frames per sec.). In addition, while high-speed networks are becoming widely used as backbones, today's installed network base will continue to be used to support communication within buildings and campuses. Thus for the foreseeable future, continuous media applications will need to be supported in the presence of delay jitter.

8.4 Future Work

The research presented in this dissertation suggests several issues that should be addressed in the future. These include issues in the areas of real-time systems, delay jitter

management policies, and overall network and operating system support for continuous media.

8.4.1 Real-time Systems

In this work, I used a real-time operating system support a continuous media application that was designed to process frames with bounded delay. Equally powerful real-time services will be necessary to support continuous media applications in general-purpose computing environments; ideally, such services could be integrated into existing operating systems. In general, the problem that must be addressed is that of ensuring that non-real-time workloads (*i.e.*, work that does not specify its performance requirements and does not receive performance guarantees) receive the best performance possible consistent with the real-time workload receiving guaranteed performance.

Another issue that has been highlighted in this research is that neither the periodic nor the sporadic model of real-time workloads capture the properties of the real-time workloads generated in an application that supports continuous media. Fundamentally, the average rate at which work must be performed is based on the frame rate (*e.g.*, frames arrive at the display workstation at an average rate equal to the frame rate). However, over short intervals, tasks are often invoked at a higher rate (*e.g.*, because of congestion, several frames arrive at the display workstation in a burst). Thus, the workload is not periodic. Furthermore, an assumption that such a workload is sporadic is extremely conservative. A new model of real-time workload is necessary.

8.4.2 Evaluation of Delay Jitter Management Policies

The most important outstanding issue in the development of delay jitter management policies is that of quality measures. In this work, I have compared policies using average display latency and average gap rate and a simple comparison rule. However, there are many other factors that can influence perceived quality including the distribution of gaps throughout a conference, the number of display latency changes, and the distribution of periods of high and low display latency throughout a conference. Policies that adapt to current conditions cannot be developed or tuned without quality measures that allow fine-grained distinctions of perceived quality.

As an example, consider the problem of choosing good threshold values for the queue monitoring policy. Simulation using a variety of threshold values indicates that large

changes in threshold values may only produce small changes in average display latency and average gap rate. As such, work on choosing threshold values will involve making tradeoffs that result in small changes to display latency and gap rate. While a simple measure of quality may be sufficient to evaluate the gross performance characteristics of threshold setting, it will not be sufficient to properly evaluate these small changes.

Another issue that should be addressed is the extent to which the queue monitoring technique scales. The study presented in this work used audio and video data transmitted over a campus-sized network. Future work should repeat the study of queue monitoring and delay jitter for a succession of larger networks. Such a study will help to identify the types of networks in which delay jitter is low enough that continuous media applications can be supported without resorting to network services with specialized support for real-time communications.

8.4.3 Network and Operating System Support for Continuous Media

The emphasis in this work has been on managing the effect of delay jitter on the display of continuous media. A related issue is that of preventing or minimizing data loss. This must be addressed in both the operating system and the network transport system. On the operating system side, this dissertation has already shown that it is possible to prevent loss through the use of real-time systems design, analysis and implementation techniques. On the network side, possibilities include traditional techniques such as timeouts with retransmission and forward error correction (FEC). However, the fact that recovering lost frames requires time implies that frames will experience greater delay jitter (presumably FEC will result in less delay jitter than the use of timeouts). As a result, the effect of error correction mechanisms on queue monitoring and other delay jitter management policies must be investigated.

Another issue that should be addressed is flow control. Throughout this work, it has been assumed that applications support continuous media that is acquired and displayed at a fixed frame rate and at a fixed resolution. Under this assumption, continuous media applications require a certain commitment of resources such as network bandwidth and processor time. Flow control mechanisms could be used to change the frame rate or the resolution in response to changes in available resources. Such mechanisms would help an application to dynamically adapt to changes in its environment. Furthermore, such mechanisms would help to alleviate transient overload conditions such as network congestion.

Overall, the fundamental question that must be addressed if continuous media is to be supported in general-purpose computing environments is: what are the service abstractions that should be provided by general-purpose operating systems and network transport systems that will effectively support both traditional data and continuous media? I believe that this dissertation has made a substantial contribution towards answering this question.

References

[1] Anderson, D.P., Tzou, S.-Y., Wahbe, R., Govindan, R., Andrews, M., 1990. *Support for Continuous Media in the DASH System*, Proc. Tenth Intl. Conf. on Distributed Computing Systems, Paris, France, (May), pp. 54-61.

[2] Anderson, D.P., Herrtwich, R.G., Schaefer, C., 1990. *SRP: A Resource Reservation Protocol for Guaranteed Performance Communication in the Internet*, University of California Berkeley, Dept. of Electrical Eng. and Computer Science Technical Report, TR-90-006, (February).

[3] Azuma, R., Bishop, G., 1994. *Improving Static and Dynamic Registration in an Optical See-through HMD*. Proceedings of SIGGRAPH '94, Orlando, FL, July 24-29, 1994, pp. 197-204.

[4] Baruah, S., Mok, A., Rosier, L., 1990. *Preemptively Scheduling Hard-Real-Time Sporadic Tasks with One Processor*, Proceedings of the Real-Time Systems Symposium, IEEE, (December), pp. 182-190.

[5] Dupuy, S., Tawbi, W., Horlait, E. 1992. *Protocols for High-Speed Multimedia Communication Networks*, Computer Communications, Vol. 15, No. 6, (July/August), pp. 349-358.

[6] Ferrari, D., 1990. *Client Requirements for Real-Time Communication Services*, IEEE Communications, (November), pp. 65-72.

[7] Ferrari, D., Banerjea, A., Zhang, H., 1992. *Network Support for Multimedia, A Discussion of the Tenet Approach*, University of California at Berkeley, TR-92-072.

[8] Ferrari, D., 1992. *Delay Jitter Control Scheme For Packet-Switching Internetworks*, Computer Communications, Vol. 15, No. 6, (Jul/Aug), pp. 367-373.

[9] Fisher, T., 1992. *Real-Time Scheduling Support in Ultrix-4.2 for Multimedia Communication*. Proc. of the Third International Workshop on Network and Operating System Support for Digital Audio and Video, San Diego, CA, November 1992, V. Rangan (ed.), Lecture Notes in Computer Science, Springer-Verlag, Vol. 712, pp. 321-327.

[10] Govindan, R., Anderson, D.P., 1991. *Scheduling and IPC Mechanisms for Continuous Media*, Proc. ACM Symp. on Operating Systems Principles, ACM Operating Systems Review, Vol. 25, No. 5, (October), pp. 68-80.

[11] Gruber, J. G., Strawczynski, L., 1985. *Subjective Effects of Variable Delay and Speech Clipping in Dynamically Managed Voice Systems*. IEEE Transactions on Communications, Vol. COM-33, (August), pp. 801-808.

[12] Harbour, M., Klein, M., Lehoczky, J., 1991. *Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority*, 12th IEEE Real-Time Systems Symp., San Antonio, TX, December 1991, pp. 116-128.

[13] Hehmann, D., Herrtwich, R.G., Shulz, W., Shütt, T., Steinmetz, R., 1992. *Implementing HeiTS*: *Architecture and Implementation Strategy of the Heidelberg High-Speed Transport System*, Proc. of the Second International Workshop on Network and Operating System Support for Digital Audio and Video, Heidelberg, Germany, November 1991, R. Herrtwich (Ed.), Lecture Notes in Computer Science, Springer-Verlag, Vol. 614, pp. 33-44.

[14] Herrtwich, R.G., Nagarajan, R., Vogt, C., 1991. *Guaranteed Performance Multimedia Communication Using ST-II Over Token Ring*. Technical Report, IBM European Networking Center.

[15] Herrtwich, R.G., Delgrossi, L., 1992. *Beyond ST-II: Fulfilling the Requirements of Multimedia Communication*, Proc. of the Third International Workshop on Network and Operating System Support for Digital Audio and Video, San Diego, CA, November 1992, V. Rangan (Ed.), Lecture Notes in Computer Science, Springer-Verlag, Vol. 712, pp. 25-31.

[16] Hopper, A., 1990. *Pandora*: *An Experimental System for Multimedia Applications*. ACM Operating Systems Review, vol. 24, no. 2, (April), pp. 19-34.

[17] Intel, 1990. *ActionMedia 750 Software Library Overview*, Intel Corporation.

[18] Intel, 1990. *ActionMedia 750 Software Library Reference*, Intel Corporation.

[19] Intel, 1993. *Intel ProShare Personal Conferencing Video System 200*. Intel Corporation.

[20] IBM, 1990. *Local Area Network Technical Reference*, IBM Corporation, 4th Ed.

[21] Issacs, E., Tang, J.C., 1993. *What Video Can and Can't Do for Collaboration*: *A Case Study*, Proc. of ACM Multimedia, pp. 199-205.

[22] Jahanian, F., Mok, A., 1986. *Safety Analysis of Timing Properties in Real-Time Systems*, IEEE Transactions on Software Engineering, Vol. SE-12, No. 9, (September), pp. 890-904.

[23] Jeffay, K., 1989. *The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations*, Ph.D. Thesis, University of Washington, Department of Computer Science, Technical Report #89-09-15.

[24] Jeffay, K., Stone, D.L., Smith, F.D., 1992. *Kernel Support for Live Digital Audio and Video*, Computer Communications, Vol. 15, No. 6, (Jul/Aug), pp. 388-395.

[25] Jeffay, K., 1992. *Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems*, Proc. 13th IEEE Real-Time Systems Symp., Phoenix, AZ, December 1992, pp. 89-99.

[26] Jeffay, K., Stone, D.L., 1993. *Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems*, Proc. 14th IEEE Real-Time Systems Symp., Raleigh-Durham, NC, December 1993, pp. 212-221.

[27] Jeffay, K., Stone, D.L., Smith, F.D., 1994. *Transport and Display Mechanisms for Multimedia Conferencing Across Packet-Switched Networks*, Computer Networks and ISDN Systems, Vol. 26, No. 10 (July), pp. 1281-1304.

[28] Jones, A., Hopper, A., 1993. *Handling Audio and Video Streams in a Distributed Environment*, Proc. ACM Symp. on Operating Systems Principles, Asheville, NC, December 1993, Operating Systems Review, Vol. 27, No. 5, pp. 231-243.

[29] Kessler, G., 1991. *Inside FDDI-II*, LAN Magazine, (March), pp. 117-125.

[30] Le Boudec, Jean-Yves, 1991. *The Asynchronous Transfer Mode: A Tutorial*, IBM Research Report RZ 2133, (May).

[31] Liu, C.L., Layland, J.W., 1973. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the ACM, Vol. 20, No. 1, (January), pp. 46-61.

[32] Luther, A.C., 1991. *Digital Video in the PC Environment*, McGraw-Hill, Second Ed.

[33] Mauthe, A., Schulz, W., Steinmetz, R., 1992. *Inside the Heidelberg Multimedia Operating System Support*: *Real-Time Processing of Continuous Media in OS/2*, IBM ENC Technical Report No. 43.9214, (September).

[34] Mercer, C., Savage, S., Tokuda, H., 1994. *Processor Capacity Reserves*: *Operating System Support for Multimedia Applications*. Proc. of the International Conference on Multimedia Computing and Systems, Boston, MA, May 14-19, 1994, IEEE Computer Society Press, pp. 90-99.

[35] Minzer, S., 1989. *Broadband ISDN and Asynchronous Transfer Mode (ATM)*, IEEE Communications, (September), pp. 17-24.

[36] Montgomery, W.A., 1983. *Techniques for Packet-Voice Syncronization*, IEEE Journal on Selected Areas in Comm., Vol. SAC-1, No. 6, (December), pp. 1022-1028.

[37] Naylor, W.E., Kleinrock, L., 1982. *Stream Traffic Communication in Packet-Switched Networks*: *Destination Buffering Considerations*, IEEE Trans. on Communications, Vol. COM-30, No. 12, (December), pp. 2527-2534.

[38] Nieh, J., Hanko, J., Northcutt, D, Wall, G., 1993. *SVR4 UNIX Scheduler Unacceptable for Multimedia Applications*. Proc. of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video, Lancaster, U.K., December 1993, D. Shepherd et al. (Eds.), Lecture Notes in Computer Science, Springer-Verlag, Vol. 846, pp. 41-53.

[39] Park, C.Y., Shaw, A.C., 1990. Experiments with a Program Timing Tool Based on Source-Level Timing Schema, Proc. of the Eleventh IEEE Real-Time Systems Symposium, Lake Buena Vista, FL, December 1990, pp. 72-81.

[40] Ramanathan, S., Rangan, P.V., 1992. *Continuous Media Synchronization in Distributed Multimedia Systems*, Proc. of the Third International Workshop on Network and Operating System Support for Digital Audio and Video, San Diego, CA, November 1992, V. Rangan (Ed.), Lecture Notes in Computer Science, Springer-Verlag, Vol. 712, pp. 328-335.

[41] Rangan, P.V., Vin, H.M., 1991. *Designing File Systems for Digital Video and Audio*, Proc. ACM Symp. on Operating Systems Principles, ACM Operating Systems Review, Vol. 25, No. 5, (October), pp. 81-94.

[42] Reed, D.P., Kanodia, R.K., 1979. *Synchronization with Eventcounts and Sequencers*, Comm. of the ACM, Vol. 22, No. 2, (February), pp. 115-123.

[43] Ross, F., 1989. *An Overview of FDDI*: *The Fiber Distributed Data Interface*, IEEE Trans. on Selected Areas in Comm., Vol. 7, No. 7, (September), pp. 1043-1051.

[44] Schulzrinne, H., 1992. *Voice Communication Across the Internet: A Network Voice Terminal*, Technical Report, Univ. of Massachusetts.

[45] Schulzrinne, H., 1993. *Issues in Designing a Transport Protocol for Audio and Video Conferences and other Multiparticipant Real-Time Applications* , Internet Engineering Task Force, Internet Draft, (October).

[46] Schulzrinne, H., Casner, S., 1993. *RTP*: *A Transport Protocol for Real-Time Applications* , Internet Engineering Task Force, Internet Draft, (October).

[47] Shankar, A.U., 1993. *Reasoning Assertionally about Real-Time Systems*, CS-TR-3047, University of Maryland.

[48] Shaw, A.C., 1989. *Reasoning About Time in Higher-Level Language Software*, IEEE Trans. on Soft. Eng., Vol. SE-15, No. 7, (July), pp. 875-889.

[49] Smith, F.D., 1991. Personal communication.

[50] Terry, D.B., Swinehart, D.C., 1988. *Managing Stored Voice in the Etherphone System*, ACM Trans. on Computer Systems, Vol. 6, No. 1, (February), pp. 3-27.

[51] Tokuda, H., Kitayama, T., 1993. *Dynamic QOS Control Based on Real-Time Threads*. Proc. of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video, Lancaster, U.K., December 1993, D. Shepherd et al. (Eds.), Lecture Notes in Computer Science, Springer-Verlag, Vol. 846, pp. 114-123.

[52] Topolocic, C., 1990. *Experimental Internet Stream Protocol, Version 2* (*ST-II*). Internet Network Working Group, RFC 1190, (October).

[53] Turner, Charles J., Peterson, Larry L., 1992. *Image Transfer*: *An End-to-End Design*, Comp. Comm. Review, Vol. 22, No. 4, (October), pp. 258-268.