Admission Control for Independently-authored Realtime Applications

by

Robert Kroeger

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2004

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis presents the LiquiMedia operating system architecture. LiquiMedia is specialized to schedule multimedia applications. Because they generate output for a human observer, multimedia applications such as video games, video conferencing and video players have both unique scheduling requirements and unique allowances: a multimedia stream must synchronize sub-streams generated for different sensory modalities within 20 milliseconds, it is not successfully segregated until it has existed for over 200 milliseconds and tolerates occasional scheduling failures.

LiquiMedia is specialized around these requirements and allowances. First, LiquiMedia synchronizes multimedia tasks by invoking them from a shared realtime timer interrupt. Second, owing to multimedia's tolerance of scheduling failures, LiquiMedia schedules tasks based on a probabilistic model of their running times. Third, LiquiMedia can infer per-task models while a user is segregating the streams that the tasks generate.

These specializations provide novel capabilities: up to 2.5 times higher utilization than RMS scheduling, use of an atomic task primitive 9.5 times more efficient than preemptive threading, and most importantly, the ability to schedule arbitrary tasks to a known probability of realtime execution without a priori knowledge of their running times.

Acknowledgements

Friends, money and academic advice got me through this thesis. Slowly. All deserve my thanks.

First, on the subject of academic advice, I thank my supervisor, Bill Cowan for his guidance, insight and particularly his patience. Also, I thank my committee for their prompt and valuable feedback.

As for money, many organizations have supported this research — I am grateful for the support of NSERC, IBM, ITRC and LiquiMedia Inc. Ironically, I am also grateful for the lack of money: the stock crash of 2001 bankrupted my salaried thesis avoidance opportunity.

I would have given up long ago without the encouragement of family and friends: many CGL members over the years, Tresidder alumni and the yawners. Of these, special thanks must go out to Ian Bell, Igor Benko, Celine Latulipe, Peter Mayo, Alex Nicolaou and Shinji Sato for (sometimes inadvertently) convincing me to continue.

Lastly, thanks to Peter Kokkovas for the totally cool name.

Trademarks

HRV, Java, JMF, JavaBeans, NeWS, VIS, TAAC-1, Solaris and SPARC are trademarks of Sun Microsystems Inc.

DirectX, DirectShow, COM, Windows NT, Windows 2000 and Vizact are trademarks of Microsoft Inc.

QuickTime, Core Image, Aqua and MacOS X are trademarks of Apple Computer Inc.

LiquiMedia, LiquiOS and MVM were trademarks of LiquiMedia Inc.

Pentium, SSE and MMX are trademarks of Intel Inc.

POSIX and UNIX are trademarks of the X/Open group.

WindRiver and VxWorks are trademarks of WindRiver Inc.

Virtuoso and Eonic are trademarks of Eonic Systems Inc.

Precise is a trademark of Precise Software Technologies.

SPARK is a trademark of Realtime Microsystems Inc..

HP, DEC, VMS and Alpha are trademarks of HP Inc.

Qualcomm and BREW are trademarks of Qualcomm Inc.

Streammaster and Motorola are trademarks of Motorola Inc.

TeraLogic is a trademark of TeraLogic Inc.

Mwave is a trademark of Texas Instruments Inc.

Chromatic is possibly a trademark of ATI Inc.

Trimedia is a trademark of Philips Inc.

SGI, REACT and IRIX are trademarks or SGI Inc.

Harmony is a trademark reserved for the Crown.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis describes an operating system architecture called LiquiMedia. I designed LiquiMedia to schedule resources for streaming multimedia applications. The LiquiMedia architecture adds a capability for admission-controlled independently-authored realtime fragments to any traditionally-scheduled operating system. The LiquiMedia scheduler allocates sufficient resources to these realtime fragments to provide realtime media processing.

An obvious question arises: what characteristics of multimedia applications differentiate them from traditional realtime applications such that they warrant specialized operating system support? A multimedia application differs from a traditional realtime application because its criteria for success is the generation of multiple *information streams* for human observers.

1.1 Information Streams

The information stream is a concept of perceptual psychology. Humans pre-attentively segregate time-varying data from all sensory modalities such as vision and hearing into streams that can then be the subject of attention [Bre90]. Successful segregation requires an information stream to have three characteristics.

First, a stream must exhibit *continuity*: it evolves at the appropriate time scale for the medium, in *realtime*. The time scale depends on the modality: film requires the continuous display of 24 frames per second while CD-quality audio require samples at 44.1kHz. Any interruption longer than a modality-specific limit breaks a stream into separate and unrelated sensory events. For example, when a CD player skips in its attempt to play a badly scratched CD, it divides what should be a single music stream into multiple non-musical sensory events.

Second, for the observer to pre-attentively group information streams originating from different sensory modalities, the streams must be appropriately synchronized. For example, to

Device	Expectation Delay
CRT warm-up	9 s
VCR playback (head spinning)	2 s
local phone call, touch tone	1.5 s from last digit to first ring

Table 1.1: Expectation delay of common consumer media devices.

avoid the appearance of a badly dubbed movie, speech audio must not precede the corresponding images of mouth movements by more than 60ms or lag the the images by more than 200ms [MS85, MGSW96].

Third, an information stream must exist for a minimum duration before an observer can segregate it from the surrounding sensory ground where this delay depends on the sensory modality perceiving it [OR86, Bre90]. For example, the segregation delay of both audio streams [Bre90, pg. 66] and video streams [OR86] is approximately 200 milliseconds.

While a sensory event shorter than the segregation delay can still be perceived, the observer does not treat it as an information stream. For example, compare a firing camera flash (event) to a laser light show (stream) or a warning beep (event) to a Mozart symphony (stream). The stream does not exist in the mind of its perceiver until it has exhibited continuous existence for at least the minimum segregation delay. Streams can therefore be categorized by whether they are younger or older than this minimum. Streams younger than the segregation delay are referred to as *pre-threshold* streams in this thesis while streams older than the segregation delay are referred to as *segregated* or *established* streams.

These three characteristics of information streams constrain the operation of a stream-generating application. For example, a software movie player produces a single information stream consisting of two sub-streams (or tracks) for two sensory modalities: vision and audition. Once the observer has segregated the movie stream, the player must continuously provide a stream of PCM audio samples at not less than 22kHz and a stream of video frames at a minimum of 20Hz.[1] Finally, the player must keep the audio and video tracks synchronized with one another to within 20 to 40 ms.[2]

In addition to the segregation delay, a user's experience with an information stream also has *expectation delay*. Figure 1.1 shows these two latencies in context. The expectation delay is the time that a user is willing to wait between activating (perhaps by pressing a button)

[1] Because human psychology determines the sampling rates needed for a satisfactory stream, exact lower bounds do not exist [Har88, Han89]. I choose these particular values because most people notice the inferiority of streams at lower sample rates.

[2] McGrath and Summerfield conclude from their study of inter-modal relations of audio and vision in speech understanding that 40 ms is an acceptable upper bound on the synchronization error between visual and auditory streams of humans speaking [MS85]. Some streams require even tighter synchronization: Hirsh and Sherrick showed that an observer can correctly determine the order of events occurring on different sensory modalities down to intervals of only 20 ms [HS61].

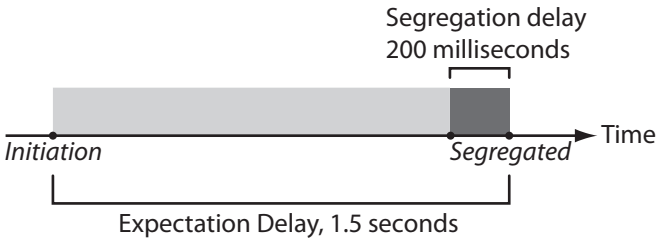

Figure 1.1: Expectation and segregation delays in context.

and perceiving an information stream. Table 1.1 lists some expectation delays of common consumer media devices — none are less than 1.5 seconds. As with segregation delay, human psychology determines the maximum acceptable expectation delay but the value is both larger at 1-5 seconds and less precise because it is a higher-level phenomenon [Shn84].

1.2 Specialized Design

LiquiMedia's architecture is specialized for stream-generating applications. It helps these applications to satisfy the psychological constraints of stream segregation while taking advantage of the segregation and expectation delays to safely schedule independently-authored code.

LiquiMedia supports stream-generating applications in three ways. First, because a segregated information stream requires a continuous realtime sample stream, LiquiMedia invokes an application's sample-generating functions at a fixed rate so that each invocation can generate a single discrete sample from the stream. Second, segregated streams must be free of interruption so LiquiMedia preferentially schedules functions generating established streams at the expense of pre-threshold streams. Third, streams bound for different sensory modalities must remain synchronized so LiquiMedia synchronizes the execution of all such functions to a single shared clock.

LiquiMedia measures a stream-generating function's running time during the segregation delay. LiquiMedia then uses these measurements to predict the function's running time. Using this prediction, LiquiMedia determines if the available processor resource permits the function to execute successfully. If so, LiquiMedia reserves the predicted processor need. Otherwise, LiquiMedia rejects the function. Because LiquiMedia makes scheduling decisions using empirical measurements, its scheduler does not have to trust application-provided running time estimates and so it can safely schedule independently-authored multimedia applications.

To define LiquiMedia's specialized architecture, I followed a strategy demonstrated by the designers of UNIX: first, define appropriate abstractions and second, design the system around these abstractions [Bac86]. LiquiMedia's fundamental abstractions consist of the *recital model*

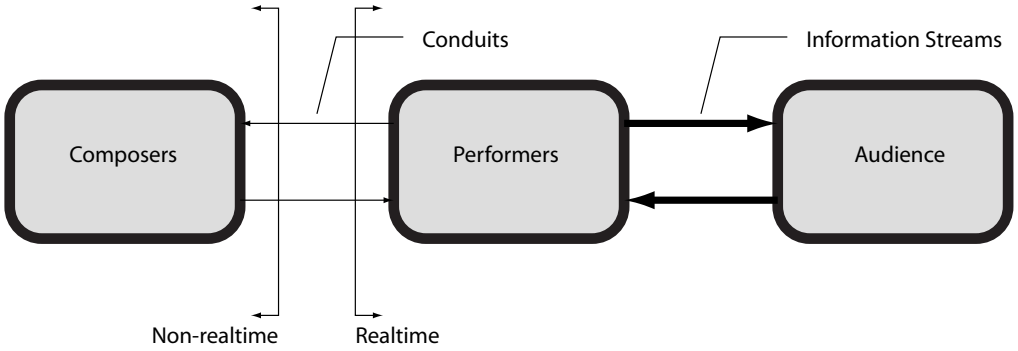

Figure 1.2: Recital Model

and the architectural requirements embodied in LiquiMedia's four design principles.

1.2.1 The Recital Model

The recital model defines the architecture of every LiquiMedia application by taking the music recital as a physical analog for a user's interaction with a multimedia application. A music recital has three participants. First, there is a composer that creates the score. Second, a number of performers interpret the score and thereby produce an information stream. Third, the audience absorbs the performance by experiencing the information stream.

As shown in Figure 1.2, information flows between the recital model's three participants. Clearly, the information flow between the performers and the audience is a synchronous realtime information stream. Conversely, information flowing from composer to performer is not realtime — it can have a latency as long as decades or centuries.

Every LiquiMedia application has this structure. Developers divide their applications into performer and composer tasks. *Composers* generate information in non-realtime. While each composer has a chance to run eventually, it is not expected to meet a precise deadline. Conversely, performers execute in realtime to produce information streams for a human audience. Finally, by further analogy with the music recital, a single *conductor* synchronizes all performers to a common clock.

1.2.2 Design Principles

The recital model defines the architecture of a multimedia application and so suggests the salient features of an operating system specialized for them: support the execution of non-realtime composers, synchronous realtime performers and provide communication between

them. However these suggestions remain too vague to guide the design of an operating system. Consequently, I augmented the recital model with four principles: the principle of partitioning, synchronous execution, modularity and ultra-fine granularity realtime. These four principles formalize a multimedia operating system's support for recital-model applications generating psychologically satisfactory streams.

1.2.2.1 Processor Partitioning

The principle of processor partitioning enables the recital model of a multimedia application. It requires application developers to divide applications into performers and composers while imposing the commensurate requirement on the operating system — that it execute these two kinds of tasks. In particular, the principle of processor partitioning demands three things: operating system support for executing composers and performers from different applications, a mechanism for them to communicate and finally an architecture that precludes the composers from ever jeopardizing the realtime execution of performers.

Why does LiquiMedia require application developers to adopt the recital model architecture for multimedia applications? The recital model and hence the principle of processor partitioning helps application developers in two ways.

First, it requires developers to consider what parts of an application have realtime constraints and which parts do not. Due consideration to this division improves an application's reliability by providing access to realtime scheduling without forcing the difficulties of realtime development on the entire application.

The alternatives are both bad: application developers can either use only non-realtime tasks or convert the entire application to run in realtime. The first choice precludes reliable generation of information streams. The second choice forces the difficulties of realtime development noted by Wirth [Wir77] on even the non-realtime portions of the application. Because realtime development is harder than non-realtime development, minimizing the amount of realtime code in an application simplifies its development.

Second, requiring developers to expose the division inside their application between realtime and non-realtime execution exposes more of an application's structure to the operating system. The LiquiMedia scheduler uses this exposed structure to better schedule the performers of all applications.

These two advantages are lost if developers reject the recital model with the excuse that its adoption is "too much effort". Consequently, the principle of processor partitioning requires the operating system to provide easy-to-use primitives for the creation, scheduling and communication between composer and performer tasks.

1.2.2.2 Synchronous Execution

The principle of synchronous execution formalizes three requirements of successful information stream generation on a computer system. First, as Abadi and Lamport [AL92] observed, a computer system generates a realtime information stream by emitting a sequence of discrete samples at a constant and sufficiently rapid rate that the samples appear continuous. Second, once the stream's age exceeds the segregation delay, the computer system must continue to generate the samples until the stream's natural end. Third, the computer system must synchronize all the generated streams with one another.

Given that performers generate all realtime information streams in the recital model, these requirements can be re-expressed more precisely. First, each invocation of a performer generates a single sample in the stream. Every *basic period*, the LiquiMedia conductor invokes all scheduled[3] performers and thereby, over a number of basic periods, generates the sequence of discrete samples that comprise a stream as described by Abadi and Lamport [AL92] and in the digital signal processing literature [OS94]. For example, the LiquiMedia prototype begins a new basic period on each video retrace interrupt — every 13ms — and hence invokes performers at a rate ideal for generating information streams for the visual modality.[4]

Second, the principle requires continuous and predictable execution: once the conductor has invoked a performer, it continues to execute that performer in all subsequent basic periods unless the performer encounters an error or requests its removal from the schedule. This "inertial" property of a performer permits it to generate psychologically continuous streams. In particular, the conductor must preserve established streams at the expense of pre-threshold streams by preferentially executing the performers generating established streams. Further, predictable execution requires that the conductor executes each application's performers in the order specified by the application.

Third, the principle requires that the conductor executes all performers from a common realtime clock and thereby synchronizes all generated streams. Moreover, this clock's period must be less than the 20 ms limit discovered by Hirsh and Sherrick to ensure that performers executed in the same basic period always appear synchronized [HS61]. While this approach guarantees synchronized streams, it does require developers to correct for possible mismatches between a stream's natural period and the conductor's basic period. For example, the duration of the LiquiMedia prototype's basic period contains a non-integral number of CD-quality audio samples so an audio-generating performer must appropriately adjust the number of samples that it produces in each invocation.

[3] This description assumes a simple schedule. Otherwise, the conductor executes all performers on a single schedule path as described in Section 4.8 and Section B.3.3.

[4] This choice of basic period does not preclude the generation of audio streams — a performer generating an audio stream produces 13ms of audio samples in each invocation. Synchronizing audio to a video clock is reasonable given the dominance of the visual modality in people [BR81].

1.2.2.3 Modularity

The principle of modularity requires LiquiMedia to safely execute multiple independently-authored applications. However the principle goes far beyond this requirement by also supporting the composition of multimedia applications from multiple independently-authored modules. In particular, the principle requires that the operating system can continue to execute correct performers in realtime despite attempting the execution of an incorrect or even malicious performer.

Three reasons justify the inclusion of the principle. First, computers have operating systems to distribute resources including, most importantly, the processor over a dynamic selection of tasks. A multimedia operating system exists for the same reason: to distribute resources between tasks that generate multiple information streams.

Second, realtime scheduling should be available to unprivileged applications. All multimedia applications, regardless of their authorship, need realtime scheduling for their performers if they are to successfully generate information streams. For example, Solaris offers both realtime and non-realtime schedulers. However because only root-privileged processes can use the realtime scheduling facilities, sensible system security policies preclude most multimedia applications such as interactive video games from using them. Similarly, cellular phones have hard realtime operating systems but force downloaded video games to execute in a non-realtime Java virtual machine. In both examples, realtime scheduling is available only to privileged applications because its misuse can easily crash the operating system.[5] The principle of modularity therefore requires that LiquiMedia can safely enforce allocations of processor to performers regardless of their origin and behaviour.

Third, the principle extends the benefits of modular object-oriented development to multimedia applications. Non-realtime software architectures such as Andrew, JavaBeans and COM, have shown that assembling applications from existing independently-authored modules greatly speeds and simplifies implementation of feature-rich applications [Bor90, Sun00, Mic00b].

The DirectShow and Java Media Framework (JMF) stream-generating frameworks based on COM and JavaBeans respectively show that the rapid development advantages of modular software extend to multimedia applications as well. However, while these frameworks support the combination of independently-authored components, neither framework executes these components in realtime [Mic00a, Jav00].[6] In combining independently-authored realtime performers in a single application, as required by the principle of modularity, LiquiMedia surpasses these modular frameworks.

[5] During LiquiMedia's development, simple programming errors regularly crashed the system.

[6] Windows 2000 provides a DirectShow facility called *kernel streaming drivers* where an information stream generating component runs at high priority inside the NT kernel [DN99]. This facility is obviously only available to privileged applications and its use entails the inconvenience of coding a device driver.

1.2.2.4 Ultra-fine Granularity

The principle of ultra-fine granularity requires that LiquiMedia can *efficiently* execute large numbers of performers such as sprite or filter functions whose execution times range from ten to a thousand microseconds. The principle is essential for two reasons. First, there will be many performers. Second, each one will be invoked many times. The principle therefore demands the smallest possible overhead on each performer invocation so as to make efficient use of processor resources.

LiquiMedia executes many performers because of the principle of modularity. The principle of modularity facilitates application development by executing applications composed of separate modules. As the example in Section 4.8 shows, even a simple video player application can contain eleven performers and modular frameworks for video decoding require more. A multi-party video conferencing application hosting four participants requires replicating the video player's eleven performers three times and adds more performers for video and audio capture. Such an application uses upward of forty performers.

LiquiMedia invokes these performers many times to satisfy the principle of synchronous execution. LiquiMedia must dispatch each performer once per basic period. At the 76Hz basic period rate of the prototype, the video conferencing application requires 3040 performer activations per second. At these rates, only the inexpensive invocations required by the principle permit efficient operation.

1.3 Applications

An implementation of LiquiMedia's four fundamental design principles has numerous practical benefits. For example, LiquiMedia's ability to execute and synchronize independently-authored multimedia applications enables the development of modular media-rich user interfaces while partitioned realtime and non-realtime tasks can reduce the hardware costs of some multimedia devices. The examples in the remainder of this section further illustrate the advantages of having the architectural features embodied in the four principles.

1.3.1 On the Desktop

Traditional desktop machines have for several years reached a level of hardware performance that supports multiprocessing time-shared operating systems such as UNIX or Windows NT. With increasing clock speeds and instruction set extensions such as MMX, streaming SIMD and VIS, desktop machines can now easily deliver sufficient operations per second to simultaneously generate many realtime information streams [Sun95a, Int99, SPE95].

However, despite abundant processor resources, scheduling failures continue to disrupt established information streams generated by traditional desktop operating systems.[7] Reasons for these failures include non-preemptible kernel code blocking the execution of stream-generating functions, overly long task quanta and virtual memory systems purging a stream-generating function's pages.[8] Further, even when a desktop operating system, such as Solaris or IRIX, actually provides realtime scheduling, it permits only specially privileged applications to execute in realtime [KSSR96, Sun95c]. These operating systems exclude un-privileged applications because they provide neither processor reservation nor a mechanism for enforcing processor reservation.

In contrast, LiquiMedia can safely schedule even untrusted independently-authored applications. Providing realtime execution to untrusted applications like audio players and games permits them to run free of scheduling glitches. Universal availability of realtime services also permits augmenting the primarily static textual content of most modern user interfaces with continuous sounds, animation and even haptic feedback.

For example, realtime fragments from different applications could cooperate to generate information-bearing sound consisting of *multiple* streams of sound. In my Master's thesis, I showed that a human user can more effectively perform a foreground task requiring the visual modality if information that the subject must simultaneously monitor is presented as streams of sound [Kro93]. Combining information-bearing sound and video streams from different applications requires an operating system capable of synchronizing multiple independently-authored realtime fragments at the latencies needed for controlling motor coordination.

Smoothly time-varying visuals have the potential to enhance an application.[9] For example, an application can generate additional brightness levels by alternating between the two closest hardware pixel values in every frame. However, generating additional brightness levels in this fashion requires reliable hard realtime [Cor70]. Further, as soon as another application tries to use this technique to display additional brightness levels, the operating system must support independently-authored realtime fragments.

Lastly, consider, a window system equipped with a force feedback pointing device. Each application both draws content on the screen outlining buttons, selection handles or text and generates small resistances as the user moves the pointing device over visible objects. Dulat showed the potential of a simulated force feedback component in improving user accuracy and speed [Dul01]. A window system making general use of this scheme needs to simultaneously execute a small force feedback information stream generator for *every* application and keep

[7] See Appendix C for a summary of a simple experiment conducted on the author's desktop machine that shows scheduling failures even with very powerful hardware and a low-utilization stream generation task.

[8] For example, traditional UNIX kernels were non-preemptible [Bac86].

[9] Note that the "Aqua" user interface for MacOS X includes significant animation elements that show a trend toward increased use of time-varying visual elements in a user interface [App00a]. Also, Microsoft has produced a program for the generation of time-varying "rich" content [Mic00c].

these streams synchronized with mouse pointer movement. Such a window system requires realtime scheduling available to independently-authored applications.

The stream-generating tasks described in the above examples all share the same three characteristics. First, each task only requires a small amount of processor time. Second, each task executes repeatedly. Third, the tasks have tight time constraints. These are the features required by the principles of modularity, synchronous execution and ultra-fine granularity, making LiquiMedia ideally suited for all of these applications.

The specialized design embodied in the four principles also enables complex interfaces consisting of multiple information streams such as the following scenario [Kro99]. As with the examples above, the applications in this example are independently-authored and contain multiple small, synchronous realtime stream-generating tasks.

> As an example, consider the experience of a consumer of Internet-provided services. Suppose he is watching a football game on television. In the dead time between plays, he uses a football interpreter/simulator which was downloaded to his media appliance, to re-run in a second window of his high definition television, portions of the already-played game, inputting different plays, blocking assignments, and pass coverage, to see what might have happened. At this point the Internet telephone rings. When he accepts the call, the avatar of an acquaintance living in New Orleans appears on the screen in a third window. He chooses an avatar to appear on the caller's display, connecting its facial expressions to his own face as interpreted from real-time captured video. The avatar's audio output will be the user's voice, and forces sensed on the user's game controller provide force-feedback on the caller's game controller, enabling virtual handshaking and back-slapping. The caller, it turns out, has noticed that our user is playing the same football simulator and wants to play in two-player mode, each second-guessing the choices of a different team and replaying the game through the simulator. Once accepting the suggestion of two way play, the user shrinks the call window to a size suitable for kibitzing and the the simulator shifts into two-player mode. Life goes on.
>
> This scenario is not an unreasonable use of a media appliance. We will examine the requirements needed to bring it to fruition. First, from the user's viewpoint, enjoying a football game in the manner discussed above requires little additional learning. A user would have to use the football simulator interface, social protocols for virtual handshaking, and so on. These are easy compared to the new habits required when users started using telephones or automobiles and will seem second nature to the people who play network games now.
>
> Second, this scenario requires immense amounts of computation and communications bandwidth. As for computational power, machines such as the Next Generation PlayStation and the even more powerful hardware that will follow have

the processing power needed. Cable modems or DSL satisfy the scenario's need for communications bandwidth.

Third, the scenario depends on rapid advances in application software. At a minimum, the software must interpret the football game and the live video of the user and his caller, simulate the complex interactions of twenty-four[10] football players and integrate input from two sources into a distributed simulation. But application software is the fastest progressing part of the computer industry and its only limitation seems to be knowing what to provide. Already, early versions of some of the required application software exist in multi-player Internet games and sports simulation games.

However, with the fourth requirement, we will see how LiquiOS [a LiquiMedia-architecture operating system] is indispensable to enabling the scenario described above. In the above scenario, a single media appliance executes a football simulator, a video player, an Internet video phone and an animated avatar. These various applications, authored in different ways by different firms must cooperate in realtime on a single system, all sharing various data sources over which they have little or no control.

This combination of software faces difficult realtime constraints. The software can tolerate some temporal faults if they result in only almost imperceptible glitches in the audio or animation while other faults such as un-synchronized audio and video will make the overall system unsaleable. Furthermore, while the system as a whole can call on the user for help in allocating resources, if it needs too much help, the system is once again unsaleable.

Only an operating system that satisfies the principles of partitioning, synchronous execution, modularity and ultra-fine realtime can enable the user interfaces discussed above. Developers can use the LiquiMedia architecture to add these features to existing desktop and embedded operating systems.

1.3.2 The Embedded Space

The architectural features required by the principle of processor partitioning can, when combined with the other principles, also enable hardware cost savings. It has become common for devices aimed at the consumer electronics market that execute a mixture of non-realtime tasks and stream-generating tasks to have separate physical processors for the two types of tasks.

While dividing tasks between processors in this way ensure that tasks from different scheduling classes cannot mutually interfere, its rigidity wastes resources when processor loadings do

[10]The game called football has significant regional difference in rules. This example assumes American football.

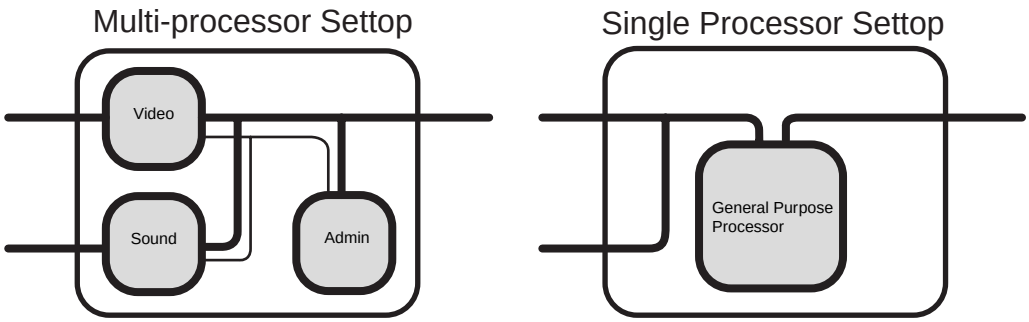

Figure 1.3: Replacing hardware partitioning with software partitioning can lower overall system costs.

not correspond to their design-time scaling. For example, some Nokia cellular phones confine all user interface processing to one of two identical processors because the second one is reserved for realtime tasks such as voice compression [Wel99]. As a result, playing a game on the phone leaves the second processor entirely idle.[11]

The Motorola Streammaster set-top box provides another example of how the fixed allocation of tasks to physical processors wastes hardware resources and hence increases device cost. The Streammaster combines a programmable video processor, a DSP and a supervisory microcontroller [Mot00]. It provides digital video decoding, gaming and web browsing functionality. These functions use considerably different proportions of the three processors in the system. For example, web browsing uses 100% of the microcontroller's processor to layout and display web pages while using perhaps 10% of the other processors. Conversely, video decoding uses 90% of the video and audio processors and only 10% of the supervisory microcontroller.

A single processor 20% faster than the Streammaster's video processor satisfies all of the Streammaster's processing needs but at a much lower parts and manufacturing cost. Figure 1.3 contrasts these two hardware architectures. However, using only a single processor to provide all of the Streammaster's various functions requires a multimedia operating system like LiquiMedia that can virtually partition a single processor between realtime stream-generating tasks and non-realtime supervisory tasks.

[11] Game play is an important function of mobile phones as it can increase average revenue per user (ARPU) [Cha04].

1.4 LiquiMedia Overview

LiquiMedia's architecture follows naturally from the recital model and the four design principles. As required by the principle or partitioning, LiquiMedia has composer and performer tasks. An IPC mechanism called a *conduit* connects these different kinds of tasks. Because composers and performers differ, separate sets of operating system primitives support their execution.

Composers are preemptible threads of computation that can be transparently suspended and restarted. In the LiquiMedia prototype, composer tasks are POSIX-style threads [Sun95b].

Performers are atomically executed functions. The conductor executes each scheduled performer once per basic period and expects the performer to voluntarily return[12] to the conductor once it has generated the current portion of an information stream. Performers are functions because [KC96] showed that the overhead of preemptive threads violates the principle of ultra-fine granularity.

The conductor executes performers from a schedule prepared by an asynchronous composer task called the *scheduler*. Having the conductor execute performers from a pre-prepared schedule has four advantages. First, it helps to satisfy the principle of synchronous realtime's "inertial" property. Second, it simplifies LiquiMedia's development by following Wirth's recommendation to minimize the quantity of code that must execute in realtime. Third, it minimizes the conductor's per-performer invocation overhead by putting schedule preparation in a composer. Finally it improves system utilization as a whole by amortizing the cost of preparing a schedule over many basic periods.

LiquiMedia computes a probabilistic estimate of each performer's running time and admission-tests performers using this estimate. The combination of admission control and a continually updated probabilistic estimate of a performer's running time permits LiquiMedia to execute independently-authored performers with a quantifiable probability of meeting realtime guarantees.

LiquiMedia has two levels of admission control: lifetime and instantaneous. The scheduler performs the lifetime context admission test as it prepares a new schedule for subsequent use by the conductor. The test consists of the scheduler verifying that an estimate of the total running of all scheduled performers is less than the basic period. The conductor invokes the instantaneous admission control test prior to invoking a performer. For each performer, it verifies that the time remaining in the basic period exceeds an estimate of the performer's running time.

Because both admission control mechanisms depend on statistical estimates of a performer's running time, they can fail. Consequently, LiquiMedia includes three techniques to address

[12] As discussed in Chapter 4, a performer that does not voluntarily return is incorrect and is removed from the schedule.

admission control failure. First, the conductor hard limits the execution of any particular performer with a watchdog timer firing at basic period boundaries. Second, the conductor uses its instantaneous context admission control mechanism to defer a performer's execution to a subsequent basic period rather than having it interrupted by the start of the next basic period. Third, the scheduler regularly updates its estimates of each scheduled performer's running time from the collected measurements of recent performer invocations and repeats the average context admission control test on the entire schedule. Should the schedule become inadmissible, the scheduler corrects it such that performers generating established information streams continue to have the specified minimum probability of executing in realtime.

LiquiMedia's admission control mechanisms require a priori knowledge of a performer's running time. LiquiMedia takes advantage of the properties of an information stream in two ways to predict a performer's running time. First, because the continuity of an information stream requires a performer to execute similar operations throughout the stream's lifetime, a performer's past running times predict its future running times. Second, the segregation delay allows the combination of the conductor and scheduler to measure a performer's running time, estimate its future running time and admission test it all before the human user segregates the newly created stream.

Consequently, a user's experience starting a LiquiMedia application such as a video player consists of invoking the player, pressing the play button, and after the expectation delay has elapsed, either seeing smooth video playback or a dialog indicating that the application cannot obtain sufficient processor for its performers. Inside LiquiMedia, the user's invocation action creates a new application as a single composer. When the user presses the play button, this composer submits a schedule of performers to the LiquiMedia scheduler.[13] The submitted performers begin executing approximately 30ms later. If sufficient processor exists for their execution, they continue until the user presses the player's stop button. Otherwise, ten basic periods later (130 milliseconds in the prototype,) the scheduler determines that the performers cannot reliably continue. It then removes them from the schedule and notifies the originating composer which communicates its regrets to the user in a dialog.

LiquiMedia's empirical approach to determining a performer's running time sacrifices the absolute guarantees of hard realtime scheduling. However, this empirical approach is a good compromise that allows LiquiMedia to achieve the principle of modularity. LiquiMedia can profile and admission-test performers within 160ms — under the segregation threshold of an information stream. Moreover, even once the performer has been admitted into the schedule based on a probabilistic estimate of its running time, a human user will tolerate the occasional glitches when it runs over its estimated running time.

Two other approaches exist for determining a performer's running time in advance: developer provided declarations and automatic determination. Neither approach can satisfy the

[13] This design choice is not ideal but simplifies the example.

principle of modularity.

Developer-provided declarations cannot satisfy the principle because, whether through machine dependency, error or malice, they cannot be trusted. For example, there is no incentive for developers to expend significant effort[14] computing an accurate bound on a performer's running time when a declaration of 0 running time will insure that the performer is not barred from the schedule for excessive resource usage. Moreover, a hardware difference, such as the processor cache size, between the developer's and user's machine can significantly change the running time of these assembly instructions. Because this approach precludes the safe execution of independently-authored performers, it cannot satisfy the principle.

Automatically determining a bound on a performer's running time cannot satisfy the principle either because the determination is impossible. Automatically bounding a performer's running time requires knowing if it can complete and is hence is an insolvable instance of the halting problem [HU79]. The alternative is to restrict a performer to a regular-expression equivalent subset of a general purpose programming language [LM95]. Given this reduction, mechanical techniques based on control-path chains, data-usage paths and integer optimization can be used to compute a worst case bound on a performer's running time. However, as with developer declarations, to execute in finite time, mechanical reduction must trust the developer to have confined his code to the acceptable language subset.

Automatic determination has at least two other practical problems. First, mechanical estimation techniques produce overly pessimistic bounds because they ignore the performance benefits of multi-level data caches, super-scalar architectures and speculative branch prediction [LHS+96, LMW96]. Second, automatic determination mechanisms require an understanding of a performer's code equivalent to a compiler's internal representation of the original source-code. Given that commercial software vendors usually ship their applications compiled, an automatic determination mechanism capable of supporting independently-authored performers therefore requires a decompiler.

1.5 Organization

The remainder of this thesis discusses LiquiMedia in greater detail. LiquiMedia is specialized for the needs of multimedia applications. It takes advantage of the domain — generating information streams for a human audience — to provide unique capabilities in a novel fashion. The four design principles of processor partitioning, synchronous execution, modularity and ultra-fine granularity embody both the specialized requirements of information stream generating applications and the unique capabilities that this specialization permits.

[14] Counting assembly language instructions.

Chapter 3 discusses numerous other operating systems and media generation frameworks whose function or approach overlaps LiquiMedia's. Owing to their number and to further clarify LiquiMedia's uniqueness of approach, I organized the systems presented in Chapter 3 into the taxonomy of operating system design features defined in Chapter 2.

Chapter 4 defines the LiquiMedia architecture in greater detail than Section 1.4 while Appendix B supplements Chapter 4 to a level of detail sufficient for reimplementation. Chapter 5 discusses the implementation of the prototype while Chapter 6 presents an empirical validation of the prototype's ability to satisfy the fundamental design principles. Finally, Chapter 7 discusses the future research suggested by the existing architecture and results.

Chapter 2

The RTOS Design Space

Chapter 1 introduced the LiquiMedia operating system for multimedia applications. LiquiMedia's specialized architecture satisfies four fundamental principles of a multimedia operating system: processor partitioning, synchronous realtime, modularity and ultra-fine granularity. LiquiMedia is unique in satisfying these principles. Furthermore, LiquiMedia satisfies the principles with a unique architecture.

Demonstrating the uniqueness of the LiquiMedia architecture requires comparing it to other realtime operating systems. However, as is demonstrated in Chapter 3, the multitude of realtime operating systems and multimedia frameworks makes this task difficult. Categorizing all of these systems by their dominant architectural features provides a structure for the exposition and facilitates comparing them to LiquiMedia. To do so, I devised a taxonomy for realtime operating systems that categorizes them by their *design aspects*. The remainder of this chapter presents this taxonomy.

The taxonomy presented here organizes operating systems by their design aspects. A design aspect is a feature of an operating system's design and is the result of an architectural choice by the OS designer. Obviously, each operating system comprises multiple design aspects and these determine its position in the taxonomy. The set of all possible design aspects form an operating system *design space*.

A design space organizes design aspects by their relationships. I adopted the three relationships defined in Sima et al. [SFK97]: "consists of", "exclusively performs", and "performs". First, the *consists of* relationship indicates that a design aspect must contain each of the orthogonal subordinate design aspects. For example the UNIX kernel consists of (among other components) a file system and a scheduler. As demonstrated by the various research projects built around Linux, operating system implementors can vary these aspects independently [www00b, www00a].

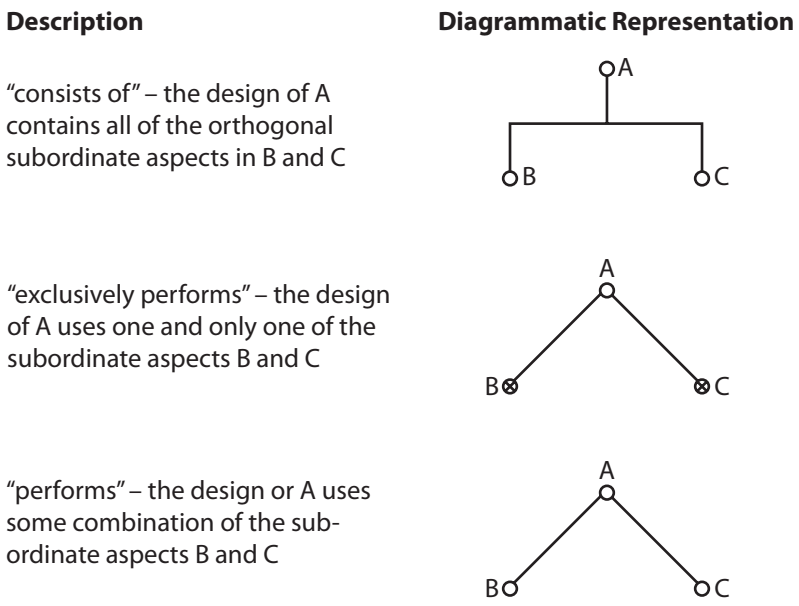

Figure 2.1: The three basic DS-Tree diagrammatic conventions.

Second, the *exclusively performs* relationship indicates that a design aspect can only be implemented by one and only one of its subordinate aspects. For example, a UNIX kernel may be single threaded (non-reentrant) or multi-threaded. Both together is not possible.

Third, the *performs* relationship indicates that a design aspect can be implemented by some combination of its subordinate aspects. For example, a UNIX kernel may implement its virtual memory subsystem with swapping, paging or both.

Sima et al. defined a diagrammatic convention called design space trees (DS-trees) for representing these relationships [SFK97]. Figure 2.1 shows the DS-tree graphical convention. DS-trees make visual the inter-dependencies between design aspects including orthogonal or exclusive relationships. DS-trees have been used extensively to illustrate design spaces [Bar76, Bar77, BS73, Bar93, BN70, BN71].

2.1 Taxonomy Overview

Because this thesis is concerned with operating system support for multimedia applications, the taxonomy presented in this chapter concentrates on those design aspects relevant to the execution of independently-authored realtime applications and ignores unrelated operating system design aspects such as name space management or storage.

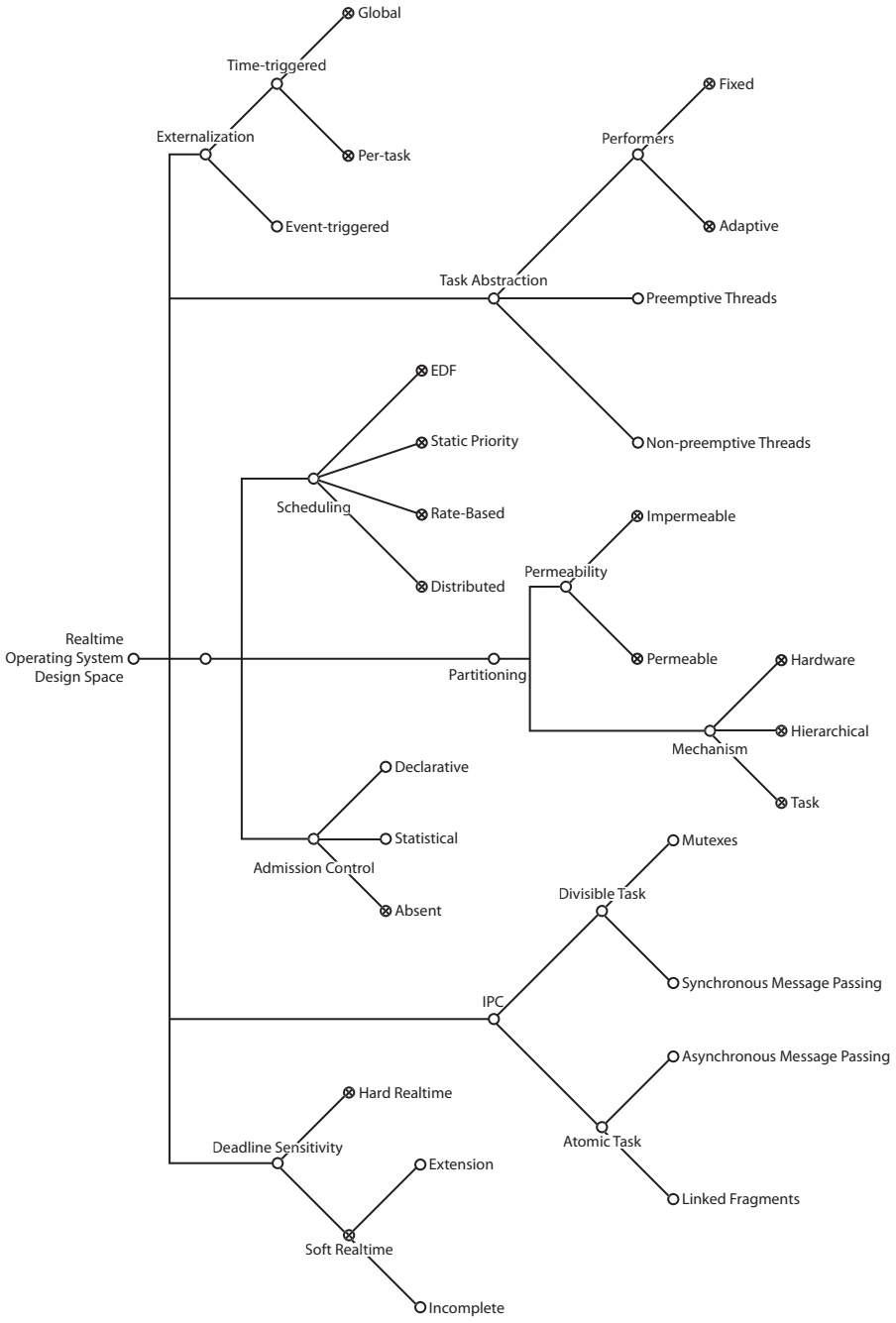

Figure 2.2: The figure shows the complete RTOS design space. An operating system consists of one (sometimes more) design aspect leaf from from each of the "consists-of" relationships in the figure.

Subject	Requires	Precludes
Hard Realtime Deadline Sensitivity	one of: Static Priority Scheduling, EDF Scheduling, Distributed Scheduling	Statistical Admission Control or Rate-based Scheduling
Static Priority Scheduling	Preemptive Threads and Time-triggered externalization	Performers and Non-preemptive Threads
EDF	Preemptive Threads	Performers and Non-preemptive Threads
Rate-based Scheduling	Preemptive Threads	Hard realtime deadline sensitivity, Performers and Non-preemptive Threads
Statistical Admission Control		Hard Realtime Deadline Sensitivity
Impermeable partitioning and Task partitioning	Preemptive Threading	Divisible Task IPC
Divisible Task IPC	Preemptive Threads or Non-preemptive Threads	Performers

Table 2.1: The table shows some relationships between different design aspects. The *Subject* column contains the subject aspects. The *Requires* column contains design aspects which the subject requires while the *Precludes* column contains aspects incompatible with the subject.

Figure 2.2 shows the complete design space. At the highest level, it contains seven top-level design aspects linked by the "consists of" relationship: task abstraction, deadline sensitivity, externalization, scheduling, partitioning, admission control and inter-process communication (IPC). The partitioning aspect contains two "consists of" sub-aspects: permeability and the partitioning mechanism.

The design space further divides each of these eight aspects into sub-aspects that have either the "exclusively performs" or "performs" relationship. Every operating system combines one leaf design aspect from each "exclusively performs" relationship and one or more design aspects from each "performs" relationship. For example, a cyclic executive like the Virtuoso

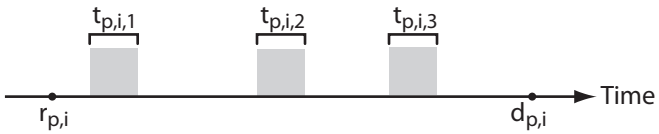

Figure 2.3: Threaded task abstractions permit sub-dividing a task's execution into multiple slices in the interval between its release and deadline.

OS for digital signal processors has global time-triggered externalization, fixed performer task-abstraction, distributed scheduling, permeable task partitioning, absent admission control, atomic IPC and hard realtime deadline sensitivity [Inc00a].

While all possible operating systems map to some combination of leaf design aspects, not all arbitrary combinations of design aspects are valid operating systems. For example, the indivisible performer task abstraction excludes the divisible task IPC aspect. Table 2.1 summarizes some of these additional constraints.

2.2 Task Abstraction

This design space assumes that an operating system supports multiple tasks and so must provide a *task abstraction*. A task abstracts a von Neumann machine: each task combines code and state and has exclusive use of the processing hardware while it is executing. The operating system controls the execution order of tasks. Typically it chooses an order that gives the outward appearance of many tasks executing simultaneously. The task abstraction has three sub-aspects with a "performs" relationship: performers, preemptive threading and non-preemptive threading. These task abstractions differ in when they permit the operating system to choose a different task to execute and the overhead required to switch between them.

Before discussing each of these sub-aspects individually, some mathematical preliminaries are necessary to formalize tasks and realtime execution. A realtime task has constraints on when it executes. Each task p has a sequence of release times and corresponding deadlines. The i-th release of task p has release $r_{p,i}$ and deadline $d_{p,i}$. Realtime execution of p requires its i-th execution happen entirely within the interval $[r_{p,i}, d_{p,i})$. The application of p determines the values of all $r_{p,i}$ and $d_{p,i}$. For example, an (un-buffered progressive scan) NTSC video player has a a sequence of releases at the start of the NTSC vertical retrace with corresponding deadlines at the end of the vertical retrace and a 33.3ms interval between successive releases.

A threaded task abstraction, by definition, permits the operating system to divide task execution into *slices*.[1] The tuple (p, i, j) uniquely identifies slice j of task p in release i. The

[1]The concept of a "slice" has no consistent name in the literature as "job", "event" and "slice" all find use. This thesis uses "slice".

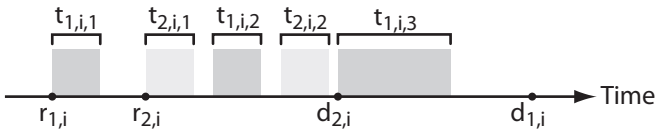

Figure 2.4: The interleaving of multiple slices of multiple tasks provides the appearance, at a sufficiently large time scale, of many tasks progressing in parallel.

operating system executes slice (p, i, j) atomically in time $t_{p,i,j}$. Figure 2.3 shows an example where the operating system has divided the execution of the i-th release of task p into three slices. In this example, p has also successfully executed in realtime because its three slices occur entirely between $r_{p,i}$ and $d_{p,i}$.

As shown in Figure 2.4, the operating system can provide the appearance of two more tasks executing simultaneously by interleaving the execution of slices comprising different tasks. Over the time interval $[r_{1,i}, d_{1,i})$, tasks 1 and 2 compute results simultaneously. The finer the granularity of the task slices, the shorter the interval over which computation appears to proceed in parallel. However, switching between slices (frequently called a context switch) itself consumes processor resources: w_f when the task abstraction permits the OS to impose the switch and w_v when context switches occur only when the task voluntarily relinquishes the processor. Necessarily, w_v is less than or equal to w_f; often w_v is much less than w_f.

2.2.1 Performers

In the performer task abstraction, there are no slices. Instead, for each release i, the operating system executes the entire task p without preemption. To insure that performers execute without being preempted, they cannot use the divisible task IPC mechanisms discussed in Section 2.8.

Atomic execution has two related benefits. First, performers have the lowest context switch overhead because context switches between performers are voluntary and occur only at their end. Second, it is trivial to implement the context switch mechanism between performers as typically an assembly language return statement suffices.

However, these benefits have a price. For the operating system to successfully interleave the execution of many applications such that they appear to execute in realtime, application developers must divide applications into performers with sufficiently short running times. Two "exclusively-performs" sub-aspects indicate the mechanism used to specify "sufficiently short" execution time: fixed and adaptive.

In the fixed execution aspect, the performer's running time is accepted as a given by the admission control mechanism (discussed in Section 2.6), which releases the performer if its

estimate of the performer's running time is less than the time available for the performer's execution. Conversely, in the adaptive execution aspect, application developers must write the performer so that it limits its running time to an upper bound computed by the admission control mechanism and provided to the performer prior to its release.

2.2.2 Preemptive Threads

A preemptive thread is a task abstraction where the operating system provides each task with the illusion of having the entire processor to itself. The operating system provides this illusion to many tasks by dividing them into slices and interleaving the execution of the slices.

Two features differentiate the preemptive thread design aspect from the other two task abstractions. First, the division of a preemptive thread p into slices is completely invisible to p except with respect to the passage of external time. Second, the operating system has complete control of the duration $t_{p,i,j}$ of each slice.

Making the boundaries between slices invisible to a preemptive thread requires a context switching mechanism that completely preserves the state of a thread at the end of each slice and substitutes the previously saved state of a different thread. The time needed for a context switch bounds the performance of an RTOS using preemptive threading.

From the internal perspective of a preemptive thread, its assembly language instructions execute sequentially without interruption. However, the operating system can insert an arbitrary delay between any two successive instructions. Consequently, a preemptive thread executes decoupled from an external clock.

This decoupling greatly simplifies the implementation of non-realtime tasks: they run until complete without consideration of the elapsed time. This feature is however inappropriate for a realtime task because it must be, by definition, bound to an external clock. Consequently, an RTOS using preemptive threads, having implemented a task abstraction that decouples task execution from external clocks, must add back in a mechanism that can synchronize different threads with external clocks. Such a mechanism both weakens the power of the preemptive thread abstraction and exposes the RTOS to the dangers of priority inversions.

2.2.3 Non-preemptive Threads

Non-preemptive threads occupy a middle ground between preemptive threads and performers. An operating system using the non-preemptive threads transparently interleaves the execution of slices of different tasks. However, the threads themselves specify at what points the operating system may divide up their execution into slices.

Requiring the task to specify these division points has two advantages. First, the operating system can use a simpler context switching mechanism when slices explicitly relinquish the

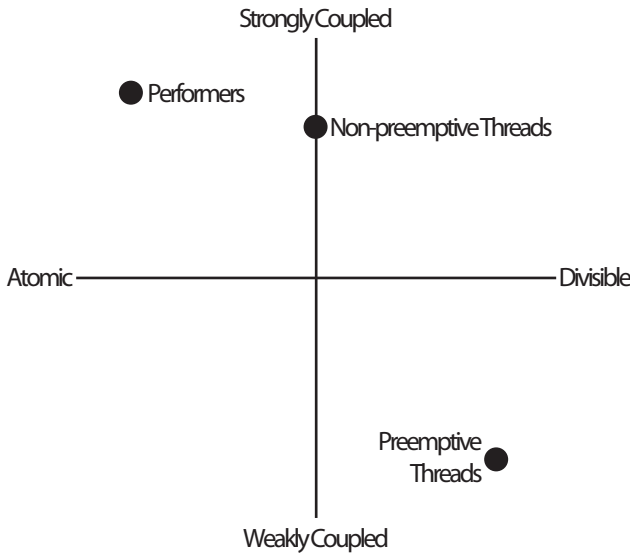

Figure 2.5: The diagram shows how the three different task abstractions differ in two axes: divisibility and coupling.

processor. Second, application developers can obtain atomic execution of a code section simply by omitting division points from the section.

However, non-preemptive threads place the operating system's ability to multi-task at the mercy of application developers because malicious code can omit division points. As a result, non-preemptive threads preclude satisfying the principle of modularity. Further, the inclusion of division points complicates the development of well-behaved applications as developers must be aware of the running time of each indivisible section.

In summary and as shown in Figure 2.5, the three different task abstractions differ in two axes: the *divisibility* of the task and its *coupling* to external time. Performers, by virtue of their atomic execution, are indivisible and totally coupled. Non-preemptive threads have some application-specified number of divisions and are coupled only within each division. Preemptive threads are arbitrarily divisible into slices and are not intrinsically coupled to realtime.

2.3 Deadline Sensitivity

Correct execution of a realtime task requires that the operating system completes execution of the task before its deadline. Formally, in a release i, the operating system must execute task p entirely within the interval $[r_{p,i}, d_{p,i})$. The values $r_{p,i}$ and $d_{p,i}$ are from a perfect realtime *external clock*. Deadline sensitivity has two "exclusively-performs" sub-aspects: hard realtime and soft realtime.

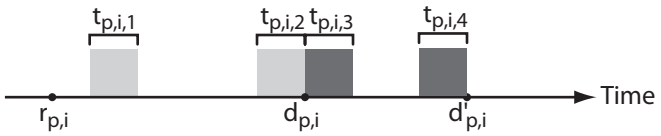

Figure 2.6: A thread that has failed to execute in realtime has one or more slices after its deadline. In the incomplete design aspect, the OS discard slices 3 and 4. In the extension design aspect, the OS extends the deadline to $d'_{p,i}$ to accommodate them.

An application that requires its tasks to complete before their deadlines in *every* release is a *hard realtime* application. An operating system capable of correctly supporting hard realtime task execution has the hard realtime deadline sensitivity design aspect.

Many realtime applications including stream generating applications do not require hard realtime deadline sensitivity because they can tolerate releases where a task fails to meet its deadlines. An operating system which occasionally fails to execute tasks in realtime can support such applications and so has the soft-realtime deadline sensitivity design aspect.[2]

A soft-realtime task might not have completed execution at its deadline. Figure 2.6 shows an example for a task p where two of its slices occur after the deadline $d_{p,i}$. The operating system's two possible responses are soft-realtime's two sub-aspects with a "performs" relationship: incomplete and extension. In the incomplete sub-aspect, the operating system still meets the deadline by not completing the task — discarding slices 3 and 4 in the example. In the extension sub-aspect, the operating system adjusts the deadline to complete the task — creating a new deadline $d'_{p,i}$ in the example.

2.4 Externalization

The externalization aspect determines an operating system's mechanism for synchronizing an application's execution with an external clock. This aspect is a necessary requirement of all realtime operating systems. A realtime operating system *must* synchronize realtime applications with events in the real world. Kopetz defined externalization's two "performs" sub-aspects: time-triggered and event-triggered operating systems [Kop90].

[2]The literature also uses the phrase "firm realtime". Formally, this is equivalent to soft realtime — some tasks may occasionally miss deadlines. Qualitatively, a "firm realtime" system is one where tasks less frequently miss deadlines than they do in a "soft realtime" system. Section 4.2 introduces a quantitative alternative: *firmness* measures the fraction of releases in which a task misses its deadline.

2.4.1 Time-Triggered

A task p in a *time-triggered* OS has a sequence of periodic releases with period T_B:

$$r_{p,1}, r_{p,1} + T_B, \ldots, r_{p,1} + iT_B, \ldots$$

and a deadline at the next release: $d_{p,i} = r_{p,i+1}$. A time-triggered operating system's key performance metric is period size: the shortest T_B that the operating system can support.

A time-triggered operating system has two subordinate design aspects with the "exclusively-performs" relationship: global period or per-task periods. In the global period design aspect, all tasks have the same period T_B. In the per-application design aspect, each task p may have an different period d'_p. A fixed global period such as found in a cyclic executive like Virtuoso simplifies the operating system and significantly reduces the need for inter-task synchronization [Inc00a]. The per-application periods found in rate monotonic operating systems like VxWorks have opposite advantages: simpler application development, more complicated operating system development and trickier synchronization [Win00].

2.4.2 Event-Triggered

An event-triggered operating system releases a task each time a previously-specified event occurs. As a consequence, an event-triggered RTOS has aperiodic release times $r_{p,1}, r_{p,2} \ldots$. The time between the i-th external event and the corresponding release $r_{p,i}$ is called the system's *latency* and is an event-triggered operating system's most important performance metric.

These two mechanisms for externalizing an operating system have complementary strengths and weaknesses. A time-triggered operating system is immune to event overload because its applications poll external sensors. Conversely, it wastes processor resources when it invokes tasks in the absence of changes in sensor values.

An event-triggered operating system eliminates this waste by invoking applications only when necessary. However, it can be overloaded by a burst of events.[3] Consequently, time-triggered operating systems are preferable when applications interact with continuously changing external sensors, generate continuously varying information streams or provide periodic output. Event-triggered operating systems are preferable when an operating system needs to efficiently handle events arriving at variable rates.

These two design aspects are not exclusive — a single RTOS can provide both externalizations. For example, hardware timer events in an event-triggered RTOS provide tasks with time-triggered externalization. Conversely, a time-triggered RTOS can poll sensors and queue tasks for execution upon discovering changes in the value of the sensors.

[3]Hence requiring one of the admission control mechanisms discussed in Section 2.6.

2.5 Scheduling

An RTOS exists to allocate resources, particularly processor resources, to tasks.[4] Consequently, the RTOS has a *scheduler* that allocates processor resources to tasks by controlling the order of task execution and the running time of task slices. A good scheduling algorithm is indispensable in building an efficient and practical operating system. Consequently, researchers have extensively explored scheduling algorithms.

Scheduling algorithms naturally sub-divide into three separate design aspects linked by a "consists of" relationship: the scheduling function, admission control and partitioning. Under this division, the *scheduling function* chooses the next task to be executed, *admission control* determines if a task can complete before its deadline and *partitioning* protects each task from errors or scheduling failures in other tasks.

These three design aspects are closely related. As a result, an actual scheduler implementation has some fixed combination of sub-aspects chosen from each of these three aspects. For example, the popular rate monotonic scheduling (RMS) algorithm has no admission control, static priority scheduling and impermeable task partitioning [LL73].

The remainder of this section discusses scheduling functions while admission control is discussed in Section 2.6 and partitioning in Section 2.7.

The scheduling function $\mathcal{S}$ takes some portion of the operating system's internal state as its arguments and returns a next task from $\mathcal{P}$. Scheduling has four sub-aspects with the "exclusively-performs" relationship: static priority, EDF, rate-based and distributed scheduling. The first three of these, static priority, earliest deadline first (EDF) and rate-based scheduling, have an algorithmic implementation of the function $\mathcal{S}$. However, $\mathcal{S}$ can also always be represented by a set of of six-tuples $\{(p, i, j; q, k, l), \ldots\}$, where an element $(p, i, j; q, k, l)$ specifies that the operating system executes slice (q, k, l) of task q at the end of slice (p, i, j).[5] If the set of tasks $\mathcal{P}$ is fixed and the operating system periodic, then the set definition of $\mathcal{S}$ is finite.[6]

When the set representation of $\mathcal{S}$ is finite, it is a practical foundation for implementing an operating system's scheduling function for three reasons. First, the implementation runs in $O(1)$ time and has a small constant. For example, a cyclic executive stores the set representation of $\mathcal{S}$ in its program structure and so can choose the next task in a single assembly language instruction. Second, an RTOS can implement any arbitrary $\mathcal{S}$ function by enumerating enough tuples. Third, it is deterministic — a valuable property in most RTOS implementations.

[4] Note however, that processor scheduling algorithms can be easily adapted to schedule other time-based resources such as bandwidth.

[5] Because this enumeration represents a function, no slice may appear on the left hand side of this mapping more than once.

[6] When finite, $\mathcal{S}$ defines a graph over slices. Consequently, graph and automata theory can be used to analyze the operation of a scheduler [ACD91a, ACD91b].

Consequently, many algorithmic schedulers use the set representation of the function $\mathcal{S}$ to improve system efficiency. For example, the BERT scheduler computes a batch of near-future tuples from $\mathcal{S}$ and places them in a FIFO [BPM99]. Then, the operating system uses the top-most tuple from the FIFO to select the next task in $O(1)$ time. Computing tuples in batches can improve efficiency through better use of hardware such as caches and better algorithms such as dynamic programming.

2.5.1 Distributed Scheduling

Distributed scheduling provides no algorithmic scheduler. Instead, the operating system "distributes" scheduling responsibility to the application: it provides the operating system with a schedule that the OS then uses to schedule the application's tasks. If the OS supports multiple applications, it must also include a mechanism to merge per-application schedules into a single system-wide schedule.

Most cyclic executives have distributed scheduling where the schedule is implicit in the application's control structures [Loc92]. The single application cyclic executive is the most efficient hard-realtime "operating system". However, embedding the schedule in the application makes debugging and alteration of the application tedious.

2.5.2 Static Priority Scheduling

In static priority scheduling, the operating system always executes the unfinished task with the highest priority. The task priority is fixed for the lifetime of the task to the inverse order statistic of the task's period — the task with the longest period has priority 0, the task with the next longest has priority 1, etc. Static priority scheduling, also known as rate monotonic scheduling (RMS) was first formalized by Liu and Layland in [LL73] and refined in [LSD89, Loc92].

Static priority scheduling provides a unique combination of features. First, it can schedule an arbitrary mixture of dynamically chosen tasks. Second, it provides well-understood deterministic behaviour. Third, it guarantees hard-realtime execution of its tasks. Fourth, an RMS scheduler is simple to implement and debug particularly if the total number of tasks is small and fixed.[7] Finally, RMS scheduling provides a perfect priority-based processor partitioning between tasks.

However, these features come at the cost of three rigid requirements. First, RMS requires preemptive threads because the operating system always executes the highest priority task

[7] An RMS RTOS can provide a compact and efficient implementation of RMS scheduling with a bit table stored in a processor register. This implementation choice is particularly appropriate for embedded applications [red00].

even if that requires suspending the execution of a lower priority task at the higher priority task's release. Second, it requires time-triggered externalization with a hardware interrupt for each task's release to provide the OS with the necessary opportunity to switch tasks. Third, as proven in [LL73], RMS can only guarantee realtime task execution if the total worst case execution time (WCET) of all n tasks over the longest (non-idle) task period remains less than

$$\lim_{n\to\infty} n(2^{1/n} - 1) \approx .693$$

of the elapsed external time.[8]

An RMS OS that provides divisible process IPC (such as mutexes) becomes susceptible to priority inversions and therefore sacrifices one of static priority scheduling's most desirable features — its hard realtime guarantee. To solve this problem, Rajkumar extended RMS scheduling with *priority inheritance*. Here, the scheduling function adjusts task priorities so that a task that holds a lock blocking the execution of a higher priority task inherits the blocked task's priority until it clears the critical section [Raj91].

However, priority inheritance mechanisms exclude hard realtime guarantees. An RMS OS guarantees hard realtime execution only if the total running time is appropriately bounded. Without priority inheritance, the total running time can easily be computed given the WCET of each *independent* task. With priority inheritance, each task's WCET must also recursively include the WCET time of every possible blocking critical section in all lower priority tasks. Such bounds are difficult to compute and overly pessimistic.

In the absence of priority inversions, RMS scheduling satisfies hard realtime constraints. A fortiore, it also satisfies soft realtime scheduling constraints. However, its utilization bound of .693 is inefficient. RMS scheduling can provide utilizations greater than .693 but only at the cost of unpredictably failing to execute lower priority tasks.

An RMS system handles overload situations poorly because it lacks an importance mechanism. Application developers set a task's *importance* to specify the seriousness of the task missing its deadline — the more serious the scheduling failure, the more important the task. Importance is logically independent of period but an RMS operating system always sacrifices long-period tasks to the benefit of short-period tasks. Thus, an overloaded RMS system has the incomplete soft realtime design aspect because it only sporadically completes long-period tasks.

2.5.3 EDF Scheduling

Liu and and Layland also defined earliest deadline first scheduling (EDF) [LL73]. When invoked at the end of any slice, an EDF scheduling function returns the runnable task with the nearest deadline.

[8] In practice, when task periods are similar, this bound can approach .88 [LSD89].

EDF scheduling has three advantages over RMS scheduling. First, the algorithm depends only on a task's releases and their respective deadlines. Consequently, unlike RMS scheduling, EDF supports both time-triggered and event-triggered externalization.

Second, provided that slices are arbitrarily small, EDF scheduling provides complete processor utilization. In practice, RTOS implementors must set a lower bound on the duration of a slice (the *quantum*) because switching between slices takes time.

Third, EDF scheduling automatically recovers from priority inversions. If a task with a near deadline blocks on another lock-holding task with a later deadline, the scheduling function returns the runnable task with the next nearest deadline after the blocked task. Eventually, the lock-holder has the nearest deadline and clears the critical section, after which all the blocked tasks will become runnable. While this recovery process may cause some tasks to miss their deadlines, it is far better than the permanent paralysis of priority inversion in RMS scheduling.

These advantages come with significant implementation cost. An EDF scheduling function maintains a table of tasks sorted by their deadline. On each release, the scheduler spends $O(\lg n)$ time updating the table of n tasks. Then, at the end of any slice, it obtains the first task from the table in $O(1)$ time.

As with RMS scheduling, EDF scheduling provides hard realtime deadline sensitivity if the tasks cannot interfere with each other. However, the EDF scheduler has much higher *jitter* than the RMS scheduler where jitter is the variability of when a task actually executes in the interval between release and deadline.

2.5.4 Rate-Based Scheduling

Rate-based scheduling (RBS) attempts to address the limitations of RMS and EDF scheduling. Preemptively-threaded tasks progress by "rate". Rate is an application-defined amount of work per unit time. Consequently, rate-based scheduling has a good fit with tasks with rate-based workloads such as video conferencing [JSS91]. Rate-based scheduling degrades gracefully under overload, handles priority inversion and has good processor utilization, but cannot provide hard realtime execution.

Because of its versatility and advantages over RMS, RBS has been the subject of considerable research. Jeffay and Goddard provide a taxonomy for the RBS design aspect [JG01]. They identify three sub-aspects of rate-based scheduling: fluid-flow scheduling, rate-based execution and the constant bandwidth server.

2.5.4.1 Fluid-flow

In fluid-flow scheduling (frequently called proportional share scheduling,) the scheduler runs tasks for fixed length quanta. At the beginning of each quantum, the fluid-flow scheduling algorithm selects the task from the set of released incomplete tasks whose execution minimizes the *lag* of all tasks.

The definition of lag requires some mathematical preliminaries. In fluid-flow scheduling, a task p requests a proportion w_p of the total processor time. (An operating system using fluid-flow scheduling incorporates admission control by testing the value of w_p prior to accepting a task into the schedule.)

The task p receives an instantaneous share of the processor:

$$f_p(t) = \frac{w_p}{\sum_{\forall x \in \mathcal{P}(t)} w_x} \tag{2.1}$$

where $\mathcal{P}(t)$ is the set of all released runnable incomplete tasks at time t. ($\mathcal{P}(t)$ is frequently called the set of *live* tasks.) A task's *service time* over the interval $[t_1, t_2)$ is then defined to be

$$S_p(t_1, t_2) = \int_{t_1}^{t_2} f_p(t)dt. \tag{2.2}$$

Equations 2.1 and 2.2 specify the behaviour of a system with arbitrarily small quanta. Because practicality requires allocating processor time in size-bounded quanta, a task has an actual service time $\overline{S}_p(t_1, t_2)$ in the same interval. The lag is then

$$\text{lag}_p(t_2) = S_p(t_1, t_2) - \overline{S}_p(t_1, t_2). \tag{2.3}$$

At the end of a quantum, a fluid-flow scheduler executes the task that minimizes the lag of all tasks. The *Earliest Eligible Virtual Deadline First* (EEVDF) scheduling algorithm insures an optimally minimal lag by using EDF scheduling on virtual task deadlines of the form $t_{p,i}/f_p(t)$ [SAWJ+96]. Here, $t_{p,i}$ is the running time of task p in release i and must be provided to the fluid-flow scheduler by the application.

Proportional share does not intrinsically provide realtime scheduling as it provides only *uniform* execution. A proportional share scheduler can schedule a task in realtime if it can satisfy two conditions. First, each task's lag never exceeds an implementation-determined upper bound. Second, a realtime task receives a share of the processor that is invariant over time.

2.5.4.2 Rate-based Execution

Rate-based execution, henceforth abbreviated RBE, is a generalization of RMS scheduling. In RMS scheduling, in any release, a task must execute completely between the release time

and the deadline. RBE scheduling relaxes this constraint by specifying a task p's realtime behaviour with a three-tuple (x, y, d'_p) where p must handle x releases in time y with relative deadline d'_p.

The actual deadline for a task p in release i is then:

$$d_{p,i} = \begin{cases} r_{p,i} + d'_p & \text{if } 1 \leq i \leq x \\ \max(r_{p,i} + d'_p, d_{p,i-x} + y) & \text{if } i > x \end{cases} . \tag{2.4}$$

RBE scheduling's use of the computed deadline $d_{p,i}$ provides two important properties. First, a task p can have up to x releases contending for the processor with the same deadline. Second, there is at least time y between release i and release $i+x$. Consequently, when releases arrive at a rate that saturates the processor, they are delayed by time y. More intuitively, the combination of these two properties provides a specified average rate of execution independent of the "burstiness" of releases.

In practice, RBE can be implemented with a modified EDF scheduler. It has excellent applicability to RTOSs featuring the event-triggered externalization design aspect.

2.5.4.3 Constant Bandwidth Server

The constant bandwidth server (CBS) is a periodic task p with a fixed period d'_p and a WCET execution time of wcet(p). As an instance of the *hierarchical partitioning* design aspect discussed in Section 2.7.2, this general-purpose task p hosts multiple event-triggered sub-tasks and the CBS scheduler.

For each release i of some sub-task q, the CBS scheduler computes q's deadline:

$$d_{q,i} = \max(r_{q,i}, d_{q,i-1}) + \frac{\text{wcet}(q)d'_p}{\text{wcet}(p)}. \tag{2.5}$$

Having computed deadlines for each sub-task, CBS schedules them using the EDF algorithm discussed in Section 2.5.3. CBS scheduling provides a load-tolerant event-triggered scheduling mechanism. but has high jitter and cannot guarantee hard-realtime execution.

2.5.4.4 Summary

These three rate-based design aspects each provide a mechanism to schedule aperiodic tasks with variable running times. Provided the running times of these tasks have known bounds, all three rate-based scheduling functions can deliver soft realtime guarantees. Further, rate-based design aspects can schedule tasks by their importance. However, rate-base scheduling has two disadvantages. First, rate-based scheduling algorithms are complex to implement. Second, they have significantly more processor (and frequently memory) overhead than RMS, EDF or distributed schedulers.

2.6 Admission Control

The scheduling algorithms described in the previous section cannot execute arbitrary collections of tasks in realtime. For example, as described in Section 2.5.2, an RMS RTOS provides realtime execution only if the sum of its tasks' WCET bounds is less than .693 of the longest task period. An operating system has an *admission control* mechanism to ensure that the set of live tasks $\mathcal{P}(t)$ always remains executable in realtime.

2.6.1 Admission Control Opportunities

An OS has two opportunities to admission test tasks before adding them to $\mathcal{P}(t)$. First, the admission control mechanism can test a task p before each release to insure that p satisfies realtime constraints before adding it to $\mathcal{P}(t)$. I call this *instantaneous* admission control because it occurs at the instant of release.

Second, the admission control mechanism can test a task when the operating system adds it to the set of realtime tasks $\mathcal{P}$. This approach insures that all live tasks satisfy realtime constraints because $\mathcal{P}(t)$ is always a subset of $\mathcal{P}$. I call this opportunity *lifetime* admission control because it persists over the lifetime of the task.

Neither admission control opportunity excludes the other. Some operating systems admission test at both opportunities because the two opportunities have different benefits.

Instantaneous admission control permits higher average processor utilizations in a soft realtime RTOS by controlling overloads. An instantaneous overload occurs when satisfying all realtime requirements requires processor utilization levels greater than 1.0. Instantaneous context admission control skips the entire release of one or more tasks so that all other tasks execute to completion. For example, an overloaded RMS RTOS partially executes many tasks. An overloaded RMS RTOS with instantaneous admission control completely skips the releases of a much smaller number of tasks so that most tasks finish successfully.

Lifetime admission control verifies that a task satisfies realtime constraints over its entire existence. An OS can use a time-consuming lifetime admission control mechanism because the mechanism's overhead is amortized over many releases.

All admission control mechanisms require estimates of tasks' execution time in every release. With this information available, admission control is simple. For example, once per-release running times are known, admission control for an RMS RTOS consists of adding up the running times of all task invocations normalized to the longest period and comparing the result to .693.

However, there is no way to mechanically generate these estimates [HU79].[9] How OS

[9]Determining *when* a task will stop executing implicitly requires determining that it *does* actually stop executing.

designers resolve this difficulty divides the admission control design space into three sub-aspects with the "performs" relationship: absent, declarative and statistical.

2.6.2 Absent Admission Control

Absent admission provides the simplest solution: the operating system has no admission control mechanism of any kind and instead assumes that all tasks will run successfully in realtime. Absent admission control is typical for embedded operating systems that support only a single application such as VxWorks [Win00]. In this case, the developers perform admission control while building the application. An operating system with absent admission control cannot safely execute independently-authored applications.

2.6.3 Declarative Admission Control

In declarative admission control, each task declares an estimate of its worst case running time (WCET) to the operating system. The operating system then admission tests the task on the basis of these estimates. Hard realtime support requires the WCET estimate. Because the operating system cannot trust the declaration of an arbitrary task, declarative admission control does not safely support independently-authored applications.

2.6.4 Statistical Admission Control

Finally, a statistical admission control mechanism admits a task if the task's probability of satisfying its realtime constraints exceeds a specified threshold. Because the task always has a finite probability of failure, statistical admission control supports only soft realtime scheduling. Statistical admission control has two sub-aspects: declared statistical admission control (DSAC) and empirical statistical admission control (ESAC).

An operating system using declared statistical admission control computes a task's probability of successful realtime execution from an application-provided probability distribution function. As with declarative admission control, this admission control mechanism does not support independently authored tasks but, as shown by SRMS, does provide utilization advantages over declared WCET estimates [AB98b].

Conversely, in ESAC, the operating system uses statistical inference to determine a task's probability of satisfying its realtime constraints from a record of its execution time in previous releases. Unlike all other admission control mechanisms, DSAC *does* support the execution of independently-authored tasks because it admission tests tasks solely on the basis of trustworthy information: the operating system's own record of the tasks' actual behaviour.

2.6.5 Mechanical Admission Control

If the various tools capable of mechanically computing a task's WCET became practical for use in an operating system, then the design space of admission control mechanisms would also require a *mechanical* sub-aspect. As it is, the mechanisms described in the literature: [LMW96, LHS+96, LM95, ACD91a, ACD91b, SKC00, Mok83] operate on only a restricted subset of a general purpose programming language, fail to incorporate the performance benefits of modern hardware such as branch prediction and require considerable resources to implement. Consequently, mechanical admission control remains impractical and unused.[10]

2.7 Processor Partitioning

For an operating system to correctly execute multiple tasks with conflicting resource needs, it requires a mechanism that prevents interference between them. Processor partitioning, sometimes called task isolation, provides the mechanism by which the operating system limits the extent to which tasks can interfere in each other's execution.

The design space of processor partitioning consists of three "performs" mechanism sub-aspects: hardware partitioning, hierarchical partitioning and task partitioning and two "exclusively performs" permeability sub-aspects: permeable or impermeable. A partitioning mechanism has the permeable sub-aspect if a scheduling or admission control failure can "leak" out of a partitioned task and prevent another task from meeting its realtime constraints. For example, a priority inversion in an RMS RTOS is a scheduling failure that has crossed the boundary between two different tasks. Permeable partitioning is less reliable but more easily implemented.

Completely permeable task-based partitioning includes operating systems that do not provide a processor partitioning mechanism. For example, the single-application Virtuoso cyclic executive provides no partitioning functionality [Inc00a].

2.7.1 Hardware Partitioning

Hardware partitioning separates tasks by executing them on separate systems where each system has a processor, private memory and I/O hardware. Running tasks on separate physical systems and providing only non-divisible IPC between them guarantees that the tasks *cannot interfere* with one another. Consequently, most systems using hardware partitioning provide impermeable partitioning by design.

[10] It can however help developers estimate a task's running time.

In hardware partitioning, each system can have a completely separate operating system with independently-selected design aspects. Consequently, system designers adjust these design aspects independently as needed by the intended application domain.

2.7.2 Hierarchical Partitioning

Hierarchical partitioning emulates hardware partitioning in software. Rather than have two or more distinct computer systems, hierarchical partitioning has virtual machines. Each virtual machine is a task in the parent operating system. As with hardware partitioning, each virtual machine has its own child operating system with possibly distinct and specialized implementations of all the design aspects including a task abstraction, scheduling function, admission control and internal partitioning mechanism. Unlike the fixed division of resources imposed by hardware partitioning, the parent operating system in hierarchical partitioning can dynamically allocate resources to each sub-operating system.

Hierarchical partitioning is an "external" approach for partitioning: the external agency of the parent operating system isolates application tasks by partitioning their hosting child operating systems. Delegating responsibility for task isolation to the parent operating system permits each child operating system to provide a specialized task abstraction, scheduling function and admission control mechanism but requires the implementation of multiple child operating systems.

2.7.3 Task Partitioning

Task partitioning takes the opposite "internal" approach: the operating system partitions individual application tasks by using these tasks themselves as the agent of isolation. This partitioning approach requires that all tasks share the same task abstraction and overloads the operating system's scheduling function and admission control mechanism to enforce the partition. The approach trades hierarchical partitioning's multiple simple sub-operating systems for one unified and more complicated operating system.

Task partitioning simplifies all application development because programmers need only learn a single task abstraction. In contrast, a hierarchical partitioning system can even further simplify application development, but only if it includes a child operating system specialized for the application's needs.

2.8 Inter-process Communication

A realtime operating system must provide inter-process communication. The inter-process communication design aspect, henceforth abbreviated IPC, permits tasks to synchronize their

operation, both with each other and with an external realtime clock.[11]

The IPC design aspect has two "performs" sub-aspects: IPC mechanisms that require divisible tasks, and IPC mechanisms that support both divisible and indivisible tasks.

2.8.1 Divisible Task IPC

The divisible task IPC design aspect includes any form of IPC where, as a result of the IPC operation, the OS may temporarily suspend the operation of the invoking task and execute a different task. Divisible task IPC excludes the performer task abstraction.[12] The design space of divisible task IPC has two subordinate design aspects with the "performs relationship": mutexes and synchronous message passing.

Mutexes serialize execution of a particular block of code but do not control the order in which tasks acquire the mutex. Instead, the scheduler chooses the order in which previously blocked tasks acquire a mutex.[13] Mutexes provide *un-ordered internal* IPC: a developer can bracket small portions of the code inside of a single task with mutex acquisition and release. Mutexes are ideal for cooperating tasks that have small amounts of shared writable data.

The un-ordered internal IPC design space also includes semaphores, monitors and multiple reader/single writer locks. All of these synchronization mechanisms limit access to code or data, can occur multiple times inside of any particular task's code and do not specify the order in which waiting tasks obtain the resource.

Conversely, synchronous message passing is *ordered external* IPC: a developer specifies control flow transfers between tasks. When a task executes a synchronous message passing primitive, it copies a datum from itself (the sender) to the recipient task and then transfers execution to the recipient. The original sender becomes schedulable after the recipient task replies. Synchronous message passing is appropriate for implementing programs featuring many different cooperating tasks with different functionality such as a heavily threaded graphical user interface [GRA89].

2.8.2 Atomic Task IPC

Atomic task IPC has two sub-aspects with the "performs relationship": asynchronous message passing and linking. These are the only two IPC design aspects that support the performer

[11] I considered calling it inter-task communications. However, the traditional acronym is "IPC" despite the fact that, at least in this case, it is actually communication between tasks. Originally I used "process" instead of "task" for the abstraction over which the operating system allocates processor resources. However, the bulk of the literature refers to this concept as a "task".

[12] Divisible IPC is compatible with non-preemptive threads because a task voluntarily invokes an IPC primitive.

[13] The application may control the order through interaction with the scheduler.

task abstraction. In asynchronous message passing, the sending task places messages in a queue. The recipient discovers the availability of a unseen message by polling the queue. Unlike synchronous message passing, asynchronous message passing never alters program control flow.

In *linking*, a task specifies the next task to execute upon completion. Linking is equivalent to the "multi-threaded" property of the Forth language [Ert01]. Linking permits a performer-based application to alter its control flow dynamically.

2.9 Summary

This chapter has presented a taxonomy for realtime operating systems that categorizes them by their *design aspects*. The design space contains seven major aspects: task abstraction, deadline sensitivity, externalization, scheduling, partitioning, admission control and IPC. Organizing the previous results from the literature by their position in this taxonomy, Chapter 3 will show that no prior OS satisfies the four fundamental principles of a multimedia operating system presented in Chapter 1.

Chapter 3

Previous Work

Chapter 1 defined four fundamental principles that a multimedia operating system must satisfy: partitioning, synchronous execution, modularity and ultra-fine granularity realtime. LiquiMedia is unique in satisfying all four principles. The remainder of this chapter supports this assertion by showing that other operating systems fail to satisfy all four principles.

Many operating systems have been developed and new ones appear regularly. Consequently, it is impractical to discuss how each system fails to satisfy all four principles. Instead, this chapter uses the design aspects presented in Chapter 2 to group operating systems into families and then describes how each family fails to satisfy one or more multimedia operating system principles.

Besides helping to demonstrate LiquiMedia's uniqueness, this taxonomy also helped guide LiquiMedia's design by revealing un-tried combinations of design aspects. Chapter 4 presents LiquiMedia's chosen design aspects and shows that they are a combination not shared with other operating systems.

3.1 Embedded Operating Systems

Signal processing applications share a similar structure — each release of a periodic task computes a single sample from a sequence of isochronous samples [OS94, AL92]. There are an enormous number of devices that contain embedded signal-processing applications. For example, all modern automobiles and cellular phones contain such applications. Many such applications have no operating system. Instead, the entire application is designed and implemented from the hardware up by a single development team. Frequently, these applications run on customized hardware — usually digital signal processors (DSPs).

However, building a signal processing application without OS support is costly so commercial RTOSs exist to reduce the application development costs. (In fact, vendors of hardware for embedded systems devote great effort to insuring that RTOSs exist for their hardware, typically by encouraging third parties to port RTOSs to them — [Tex00] for example.) I call an RTOS optimized to support a single static application an *embedded* RTOS.

An embedded RTOS supports a single static application with the smallest possible memory use. Consequently, an embedded RTOS lacks admission control and has permeable task partitioning because excluding these features minimizes the operating system's memory footprint. Instead, application implementors perform admission control at design time. All embedded RTOSs also have the hard realtime deadline sensitivity, and, to support signal processing applications, time-triggered externalization.

The embedded RTOS family has two branches: cyclic executives and embedded RMS RTOSs. The branches vary in their scheduling and task abstraction design aspects. Figure 3.1 shows these two branches. Locke provides a useful comparison between them [Loc92].

3.1.1 Cyclic Executives

The cyclic executive is a loop around one or more performers, executing each one in a fixed order at a fixed rate. This single loop provides its fixed performers with global time-triggered externalization but provides no partitioning or admission control. IPC must be atomic because the performer-style tasks are indivisible. The schedule is embedded in the structure of the code and so is distributed. Finally, the cyclic executive provides hard realtime execution.

This combination of design choices has several advantages. First, it provides the maximum hard realtime performance. Second, a cyclic executive also provides a quantifiable level of jitter. Finally, a cyclic executive has almost no operating system overhead. However, it has several corresponding disadvantages. First, it requires application developers to ensure that every execution path through a task executes in the same amount of time. Second, modifying a task's implementation may force re-writing all the other tasks to maintain a desired cycle interval. Lastly, distributed scheduling requires application designers to manually specify a fixed order of performer execution.

Perhaps because of their simplicity, cyclic executives have received no research attention since Manacher provided all the necessary theory for the design-time construction of a cyclic executive's distributed schedule in 1967 [Man67]. However, a number of commercial cyclic executive RTOS toolkits exist. Most of these toolkit operating systems provide re-usable libraries of services that an application developer statically links into a fixed-function DSP-based device. For example, SPARK, Precise/MQX and Virtuoso are examples where the OS is optimized for media processing applications on DSPs [Mic00d, Inc00b, Inc00a]. Each

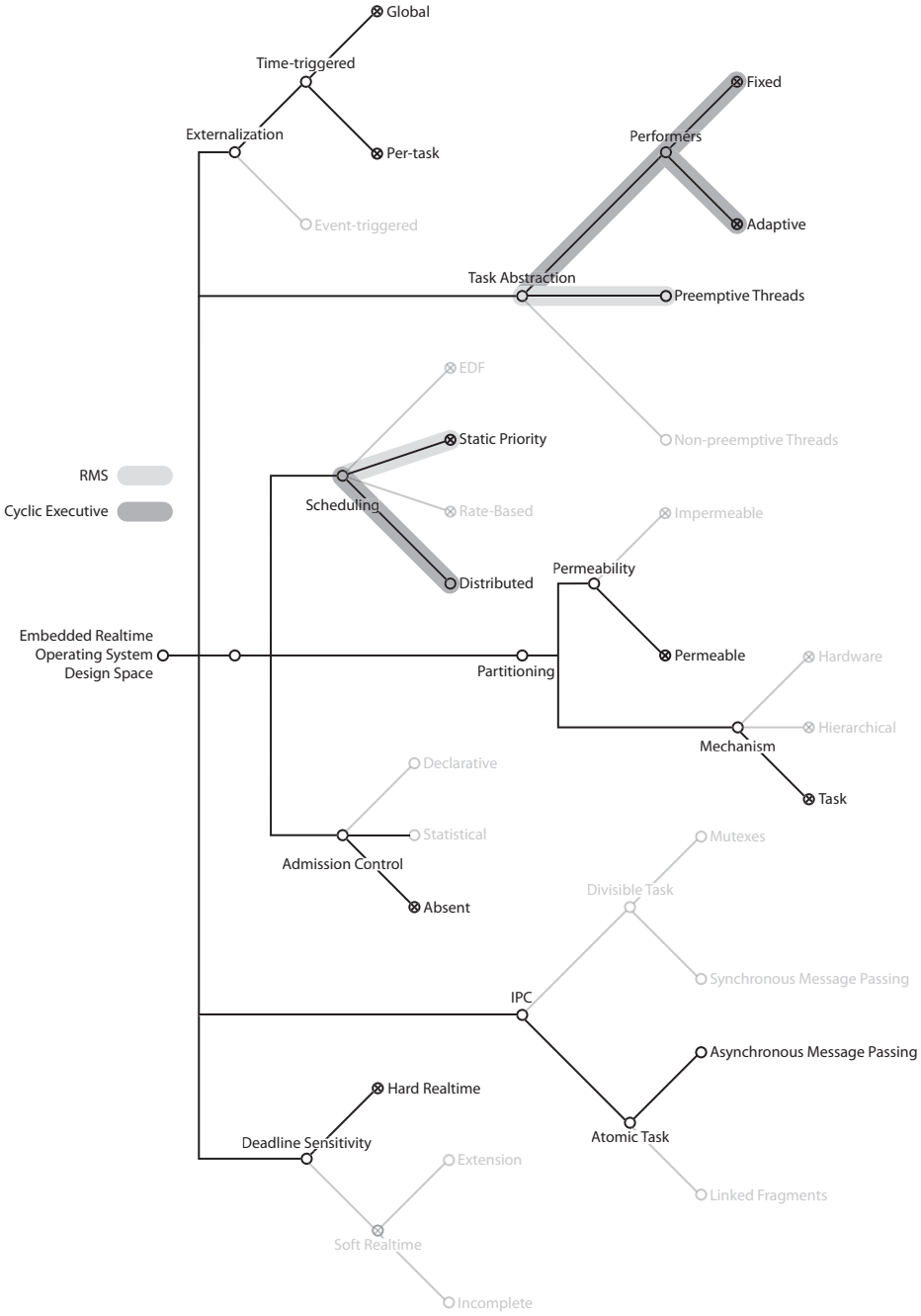

Figure 3.1: Design aspects of the embedded RTOS family including RMS operating systems and cyclic executives. Aspects not present in the family are grayed-out.

provides a framework for the development of realtime signal processing applications. The eCOS operating system toolkit can also be configured as a cyclic executive. [red00].[1]

The cyclic executive's time-triggered externalization, distributed scheduling and performer task abstraction satisfy the principles of synchronous realtime execution and ultra-fine granularity realtime. However, embedding the schedule into the application structure and the cross-task tweaking needed to keep a group of tasks running at a specified periodicity makes it impossible to satisfy either the principle of partitioning or the principle of modularity.

3.1.2 RMS Operating Systems

The embedded RMS operating systems combines, by definition, the following design aspects: per-task time-triggered externalization, preemptive task abstraction, RMS (or static priority scheduling,) task partitioning, no admission control, some form of IPC and hard realtime deadline sensitivity. An embedded RMS operating system may offer both atomic and divisible task IPC features. However, if application developers take advantage of divisible task IPC, the resultant priority inversion problem renders the RMS operating system permeably partitioned and hence only soft realtime capable. With only atomic task IPC, the RMS operating system provides impermeable task partitioning.

Impermeable task partitioning and per-task time-triggering permit different tasks to be developed mostly[2] independently. Consequently, RMS operating systems greatly facilitate team development of a single hard realtime application such as an aircraft flight controller. Further, as pointed out by Locke, unlike in the case of the cyclic executive, the RMS operating system itself is independent of the application that it supports [Loc92]. Consequently, all major commercial embedded realtime operating systems are RMS operating systems. For example, VxWorks [Win00], OS9 [Mic00e], eCOS [red00], VRTX [Men02], QNX [QNX00], Harmony [GMSW89] and LynxOS [Lyn02] are all RMS operating systems based on the scheduling foundations defined by Liu and Layland [LL73].

An RMS RTOS's combination of time-triggered externalization, RMS scheduling and preemptive threads satisfies the principles of synchronous realtime and processor partitioning but its use of preemptive threading may impede satisfying the principle of ultra-fine granularity. Further, despite support for a changing mixture of tasks, an RMS RTOS cannot satisfy the principle of modularity. Lacking an admission control mechanism, an RMS RTOS can only execute tasks in realtime if application developers have verified in advance that the

[1] eCOS is a toolkit of operating system components — it can be configured for many different sets of design space choices. One could offer an interesting realtime course based on a design space approach where the students use eCOS to build operating systems occupying different locations in the design space.

[2] Each task can be written independently, but verifying that the combination of these tasks does not violate the utilization limits of RMS scheduling cannot be done independently.

processor usage of all possible combinations of tasks satisfy the constraints discussed in Section 2.5.2. That this verification must test all possible schedulable tasks excludes executing independently-authored tasks and so cannot satisfy the principle of modularity.

Both the cyclic executive and the embedded RMS RTOS fail to satisfy the principle of modularity. Given that embedded applications do not execute independently-authored code, this failure does not reduce their utility. However, it does show that embedded operating systems cannot serve as multimedia operating systems.

3.2 Hardware Partitioning Operating Systems

Some embedded systems must implement tasks with contradictory scheduling requirements such as both time-shared and hard realtime tasks. Many computer systems provide such a capability with hardware partitioning — a separate physical processor for each scheduling domain.

Examples of complete systems that use hardware partitioning include Motorola's Streamer's [Mot00], cellular phones based on Motorola's DSP56651 [Mot98], Qualcomm's BREW [Qua01], cellular phones based on its multi-core CDMA chipsets and PC-hosted digital television systems based on the TeraLogic TL880 chipset [Mot00, Ter01]. Many other examples exist as this is the most commonly used solution for commercial systems that need to execute both hard realtime and soft or non-realtime tasks. The popularity of the hardware partitioning approach is sufficiently popular to have attracted significant tool development efforts [Wol94, QHG99, Axe, San03]. These systems use hardware partitioning to ensure that soft realtime execution cannot interfere with hard realtime execution.

Systems of this type have at least one processor that runs either an RMS or cyclic executive style embedded RTOS. However these systems have one or more additional processors that execute a different operating system. The design aspects of the complete system are the union of the design aspects of the operating systems running on each of the different processors. For example, all of [Mot00, Qua01, Ter01] have two processors: a DSP-style processor running a hard realtime cyclic executive and a more traditional processor running a general purpose operating system. In these examples, the second operating system (OS-9, BREW and Windows respectively) provides soft or non-realtime execution [Mic00e, Qua01, Mic04]. The two processors communicate via dedicated hardware such as a dual-ported memory or a pair of FIFOs.

Hardware partitioning provides a completely impermeable partition between tasks on different processors. Consequently, developers can implement each processor's tasks independently but at the cost of more complex hardware.[3]

[3]At least in the context of hardware such as set-top boxes or cellular phones, I suspect that an engineer-

The examples of hardware partitioning discussed above have two sources of inflexibility. First, they fix the resource allocation between physical processors at the time of designing the hardware. Second, they all use a fixed-function hard realtime cyclic executive on the realtime-dedicated processor. The first inflexibility is an intrinsic property of hardware partitioning and can only be remedied with one of the software partitioning schemes discussed below. Driven by a need for versatility, many realtime systems require the ability to execute a changing mix of hard realtime signal processing applications on the realtime-specific processor. The remainder of this section discusses a solution: an *extensible cyclic executive*.

An extensible cyclic executives augments a cyclic executive with a functional scheduler and an admission control mechanism. A fixed-function cyclic executive is merely a set of re-usable libraries that an application developer statically links into the application and is just barely an OS. However, an extensible cyclic executive is a true, albeit small, OS — it manages the resource needs of multiple dynamically chosen tasks. With the inclusion of admission control, an extensible cyclic executive has the following design choices: time-triggered externalization, performer task abstraction, distributed scheduling, hardware partitioning between it and the host processor and permeable task partitioning internally, atomic task IPC and declarative admission control. Figure 3.2 shows the design aspects of such an operating system.

The DSP-like devices known as "media processors" such as the HRV [NK91, HBJS91], Mwave [Tex92] and Trimedia processors [Sla97] are relevant examples because their combination of hardware partitioning and an extensible cyclic executive has the same purpose as LiquiMedia: a system optimized for the generation of information streams. Of these, the Mwave is described in more detail below because [Tex92] contained the most complete architectural description.[4]

The MWave was designed to be attached to a PC as an intelligent co-processor for time-varying media. Its system software consisted of two components: an admission control framework and scheduling package running on the PC host and a cyclic executive running on the Mwave's DSP.

The PC-hosted scheduler packed performers into one of three round-robin scheduling tables based on a task's declared performance statistics and desired execution rate. The scheduler accepts additional tasks for the Mwave so long as sufficient unallocated time in the schedule remains to hold the new task. Tasks must declare a worse-case upper bound on their processor needs.

A hardware interval timer provided time-triggered externalization at a different rate for each of the scheduling tables. The dispatch frequency of the different schedules reflected the

ing organization's most economical choice in this trade-off depends primarily on the proportion of electrical engineers and computer scientists in the organization. Members of each discipline will prefer the additional complexity in the domain that they find most comfortable.

[4]In some ways, the Chromatic media accelerator makes a better example but as the company (and consequently the white papers on its web site) has ceased to exist, I cannot provide a citation for its features.

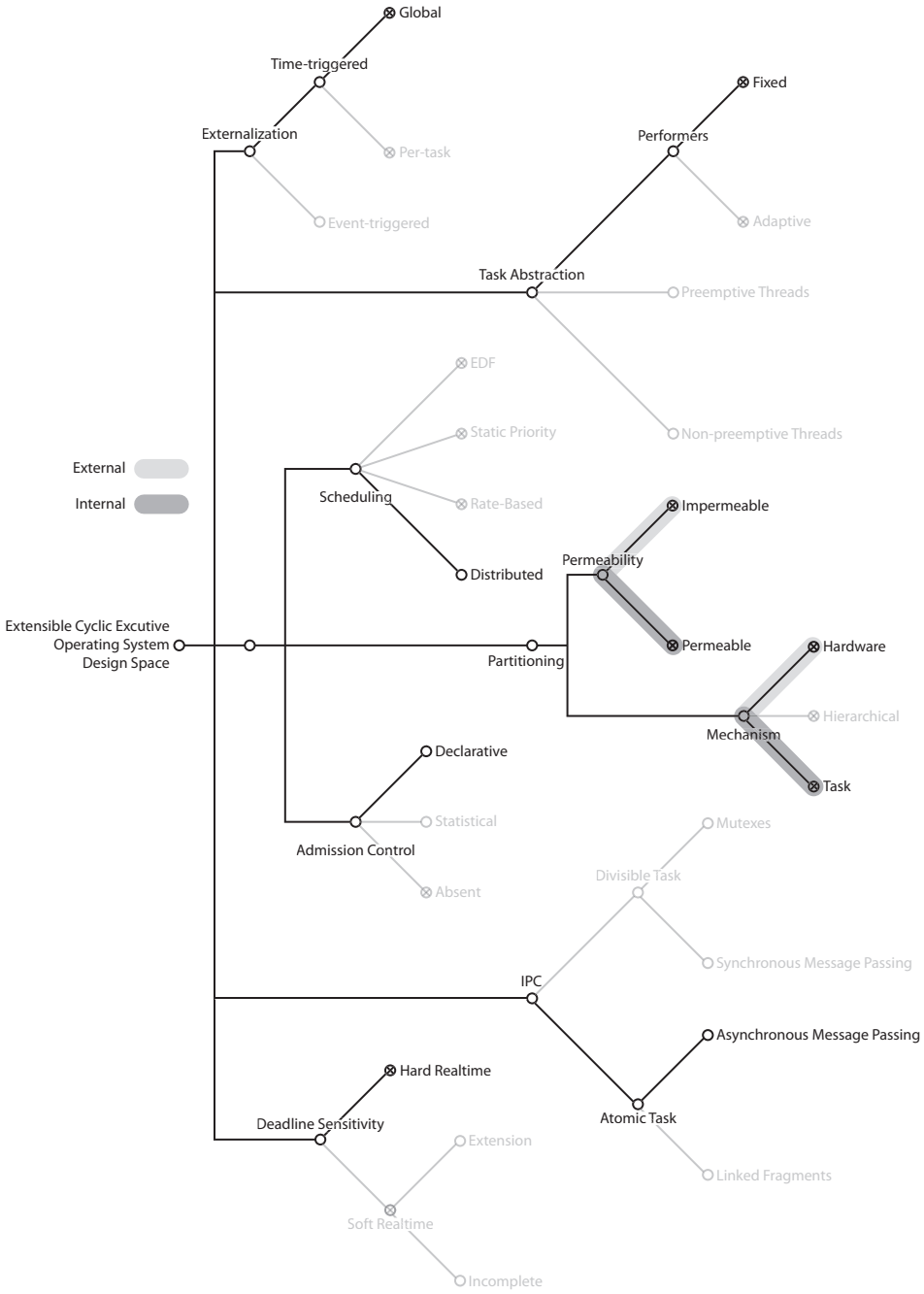

Figure 3.2: The diagram shows the design aspects of the extensible cyclic executive OS family. Aspects not present in the family are grayed-out.

intended applications of the processor: joystick and MIDI support, software modem emulation and music synthesis. Tasks dispatched at higher frequency interrupt tasks dispatched at lower frequencies. As a result, some of the tasks on the Mwave do not execute atomically. Nevertheless, these are performer-style tasks because the Mwave's operating system never interleaves the execution of tasks at a single dispatch rate.

The Mwave provided asynchronous message passing IPC via dual-ported hardware ring buffers. These permitted communication between tasks on the Mwave itself and between the host processor and the Mwave. In summary, the Mwave has an extensible cyclic executive operating system with the following design choices: global time-triggered externalization, fixed performer task abstraction, distributed scheduling, task partitioning and atomic task IPC.

Sun's High Resolution Video workstation (HRV), combined a workstation with a programmable video processing board built out of multiple i860 processors. The HRV was the successor to Sun's TAAC-1 board, that though not marketed as such, is another example of a hardware-partitioned media processor. The video board executed a custom RTOS optimized for media processing. The HRV communicated both internally and with its non-realtime SunOS host via asynchronous message passing over a *conduit*. Conduits included a rate-balancing abstraction that permitted realtime and non-realtime processing to remain synchronized [NK91].

SGI's REACT realtime extension to IRIX provides another example of a hardware partitioned extensible cyclic executive. As discussed below in Section 3.4, REACT provides distributed scheduling of hard realtime performers. To insure realtime operation even while sharing system resources with IRIX, one or more physical processors must be exclusively dedicated to realtime REACT tasks [CT00].

Hardware partitioning systems such as HRV and Mwave were purpose-built to provide multimedia operating system functionality and so satisfy the principles of partitioning, synchronous realtime and ultra-fine granularity realtime. However, these systems only permit privileged applications to submit and execute tasks on the dedicated realtime hardware. Consequently, they do not support the execution of arbitrary independently-authored realtime and do not satisfy the principle of modularity.

3.3 Media Generation Frameworks

Multimedia stream generation, particularly the playing of digitally compressed music and video, is by far the most common realtime application on general purpose operating systems.[5] Consequently, there are a number of *media generation frameworks* that attempt to provide

[5]It's the only one actually shipped with all major general purpose operating systems.

a virtual signal-processing co-processor for the generation of multimedia streams. Each media generation framework (MGF) has the same function and the same design aspects as the RTOS running on a hardware media processor such as the Mwave. A MGF has time-triggered externalization, the performer task abstraction, permeable task partitioning, declarative (or absent) admission control, distributed scheduling and atomic task IPC.

Apple's QuickTime, Microsoft's DirectShow and the Java Media Framework are the best known commercial MGF examples [App00b, Mic00a, Mic04, Jav00].[6] A sizable number of other frameworks exist including GnomeMedia, MET++, and GStreamer [Lee01, Tay01, AEW01]. QuickTime is the seminal example of a MGF and has inspired the architecture of the others.[7] Consequently, the following focuses on QuickTime.

In all of these examples, the realtime behaviour of the MGF is at the mercy of the underlying general purpose operating system. The Java Media Framework (JMF), is at the mercy of two: the "operating system" of the Java virtual machine and whatever operating system lies underneath it. Clearly the MGF operates best as a client RTOS of a parent OS that has the impermeable hierarchical partitioning design aspect. Windows 2000 attempts to provide a form of hierarchical partitioning for its native MGFs and this is discussed further below.

QuickTime is a multifaceted product. It contains a partitioning framework, an admission control mechanism, a scheduler, an IPC mechanism optimized for communications between tasks performing media processing and a container-based file-format [Hod94, Wan95, App00b]. The QuickTime container format or "movie" provides an arbitrary hierarchy of "media" objects that can correspond to the hierarchy of substreams comprising an information stream. Each media object references some underlying block of data, either on local storage, or, in a recent development, as RTP streaming content. For example, a movie is a track of sound and a track of images, where the sound may itself be comprised of multiple separate instruments and the images may consist of some arbitrary combination of text, sprites, realtime animation and compressed image sequences. QuickTime's media model is sufficiently general to encompass the entire display capabilities of modern computers as compared to the fixed capabilities of the MPEG-1 format [Tek95].

An application that wishes to use the QuickTime framework must begin by building a data flow graph of *components*. Once QuickTime begins processing media data, it repeatedly invokes the components at their desired invocation rates and executes them atomically each time. Components provide a performer-style task abstraction.

QuickTime does not provide an admission control framework for its components. Instead, QuickTime components are adaptive-style performers. A correctly written QuickTime component must adapt its processing internally to keep its execution time below the upper bound

[6] Microsoft continues to radically alter DirectShow with each release. Consequently, it is difficult to accurately determine its current abilities.

[7] Apple filed suit against Microsoft on the grounds that DirectShow borrowed too much of its architecture from QuickTime [App95].

permitted by the schedule. QuickTime defines interfaces by which the framework informs a component of the currently acceptable maximum running time. For example, consider a QuickTime video decompression component. On a fast machine, the component generates a full frame thirty times per second. On a slower machine, the video playback remains synchronized to the audio track but each invocation of the component generates only 1/6 of a frame — resulting in a frame-rate of only five frames per second. A human audience's sensory modalities impose different timeliness requirements on the various tasks comprising a multimedia application. The QuickTime framework integrates an importance mechanism that attempts to respect them.

The data flow graph provides inter-component IPC. Components operate on buffers of media information. Components invoke QuickTime functions to retrieve data buffers from upstream components and to write processed buffers to downstream components. QuickTime includes several functions for buffer management. Sample buffers are equivalent to messages and so constitute an instance of asynchronous message passing.

At the time of constructing the dataflow graph, the application must also specify a timebase for each component. The QuickTime framework dispatches each component at the specified rate. At the time of instantiation, QuickTime prepares a table that schedules component execution.[8]

QuickTime permits every component to have a separate time base. QuickTime contains an embedded time base component that takes care of obtaining the actual external time from a hardware clock and scaling and shifting this as required for each component. One of QuickTime's foremost strengths as a framework for the generation of multimedia streams is its ability to generate them synchronized to external time. Clearly, this is per-application time-triggered externalization.

QuickTime components execute in a permeable hierarchical partition as each application is responsible for making function calls into QuickTime to give the framework time to execute media processing components. Requiring cooperative applications is QuickTime's most significant weakness as a framework for signal processing — not only is there no enforced explicit allocation of resources to the media processing components, but the scheduling of what should be realtime components is not even exposed to the operating system. For example, an application may inadvertently starve its own realtime components by lingering in its non-realtime code.

Originally, Apple developed QuickTime for a a non-preemptively scheduled version of MacOS [App88]. Consequently, a correct application could guarantee realtime operation of its own QuickTime components by never letting other applications run. MacOS X has eliminated this "feature" by being based on the Mach microkernel. Despite Mach's inclusion of static priority

[8]The documentation does not exactly specify how QuickTime creates its schedule. I believe that this is deliberate.

scheduling, the version of Mach found in MacOS X is not an RTOS [App97, Raj00]. QuickTime's poor performance on MacOS X demonstrates the importance of running a MGF inside an impermeable partition on an RTOS [App00a]. However, experience with MacOS X 10.3 suggests that Apple has significantly improved QuickTime's realtime performance. Core Image coupled with programmable GPUs will add hardware partitioning to QuickTime [App04]. QuickTime also executes on Microsoft WindowsNT where it continues to exhibit performance problems caused by its lack of partitioning.

Microsoft's DirectShow provides similar functionality to QuickTime. The ASF and AVI file formats correspond to the QuickTime container file format. DirectShow permits applications to build "filter graphs". The filter is an adaptive-style performer equivalent to the QuickTime component. The DirectShow library contains a wide assortment of pre-defined filters. The filter graph provides a framework for asynchronous message passing IPC between the filter instances comprising any one graph. Applications may provide their own filters. In DirectShow, the filter graph also distributes a single clock to all the filters thereby providing DirectShow filters with global time-triggered externalization.[9] Filters drive adaptation by distributing *quality messages* which instruct upstream filters to either reduce or increase the quantity of data produced.

Unlike QuickTime, DirectShow attempts to provide a more impermeable partitioning between filters and non-realtime applications. When running on Windows 2000, selected filters are actually stubs that control filters executing in the context of the operating system kernel. These *kernel streaming drivers* execute at a priority that protects them from preemption by non-realtime applications [Sol98]. However, kernel streaming filters are not truly realtime as the kernel's processing of I/O requests can still preempt them in un-predictable ways [DN99].[10] Non-privileged applications cannot install a kernel streaming driver.[11]

The Java Media Framework (JMF) also provides a synchronous framework similar to QuickTime or DirectShow [Jav00]. An application uses the JMF just as a QuickTime application uses the QuickTime API.[12] An application first instantiates fixed performer-style media processing components and connects them together into a filter graph. The filter graph provides a framework for asynchronous message passing IPC. A clock base component provides the various JMF components with per-application time-triggered externalization. Through the Java language's support for interfaces, the JMF makes it possible to easily add new media

[9] Two different Windows applications could have different clocks if they succeeded in using DirectShow simultaneously. In practice, I have observed that each DirectShow filter tends to take exclusive ownership of any associated hardware resources.

[10] The existence of Rialto (discussed below in section 3.6.2) suggests that Microsoft will eventually attempt to correct this limitation.

[11] Information on the development of kernel streaming drivers is difficult to obtain.

[12] Apple's QuickTime development staff seemed quite contemptuous of the JMF — describing it as an incomplete copy [Nar99]. I agree with this assessment: JMF is, for all practical purposes, a functional subset of QuickTime. However, on the Win32 platform, JMF is easier to develop with.

processing components.

Like QuickTime, the JMF provides no admission control. Unlike QuickTime or DirectShow, the JMF framework does not specify a running time to its performers so they could consume all available processor resource. Fortunately, JMF does partition its performers from other code by invoking them as dedicated threads. This partition is, however, extremely permeable as Java threads lack processor reservation and exhibit considerable scheduling variance caused by garbage collection and an imprecise scheduling model.[13]

Of existing research partitioning frameworks, GStreamer appears to be the most complete [Tay01, LT96, FM01]. The designers of GStreamer copied most of its architecture from DirectShow. Most of the changes were practical improvements. GStreamer has fixed performer-style tasks, a connected graph of components, a reasonably rich library of components and fully supports execution of user-provided components. The IPC mechanism continues to be asynchronous message passing between atomic tasks, but appears from inspection to be much more efficient.[14]

All four MGFs discussed above were designed for multimedia stream generation, particularly digital music and video playback. Each framework has time-triggered externalization, the performer task abstraction possibly with adaptive execution time specification, permeable hierarchical partitioning externally and permeable task partitioning internally, little or no admission control, distributed scheduling and asynchronous message passing IPC. These frameworks all attempt to provide a software emulation of the combination of a DSP and a general purpose processor (c.f. 3.2) by embedding a DSP-like execution environment inside a general purpose operating system. However, all of the discussed frameworks fail to satisfy any of the principles of a multimedia operating system because none of the MGFs actually execute in realtime and all four principles require *realtime* execution of stream-generating code.

3.4 Distributed Scheduling Operating Systems

A wide variety of general purpose realtime operating systems exist. I categorized these RTOSs into families by their scheduling and partitioning mechanisms. Distributed scheduling operating systems combine time-triggered externalization, performer-style tasks, (permeable) task partitioning, distributed scheduling and declarative admission control. With only declarative admission control, this RTOS family fails to satisfy the principle of modularity. Relevant examples include Spring because it uses distributed scheduling in a manner similar to LiquiMe-

[13] See Appendix D for additional information on the JMF and how some of its limitations can be rectified.

[14] Application-provided filters must link against DirectShow COM objects to have access to media data. Component developers for GStreamer compile media data access macros into their components as GStreamer is available as source.

dia [SR91] and MediaVMS [VRT96] and REACT [CT00, VRT96] because both have realtime capabilities intended for information stream generating applications.

Spring[15] is a hard realtime RTOS with per-task time-triggered externalization, performer tasks, distributed scheduling and declarative admission control. Given each task's desired dispatch rate and its WCET, the Spring "planning scheduler" verifies that it can successfully interleave the execution of the tasks and, if so, constructs a fully enumerated scheduling function. Spring has distributed scheduling because the scheduling function is constructed by an external agent — albeit one provided by the operating system.

Spring is a distributed RTOS as the operating system manages resource allocations on a network of homogeneous processors. Spring uses both hierarchical and hardware partitioning to separate different classes of tasks. In particular, Spring supports event-triggered tasks by executing them inside a partition created from a time-triggered task. The Spring kernel schedules I/O and memory resources in addition to processor resources. Processor nodes communicate via asynchronous message passing [SR91]. With only declarative admission control, Spring can only support independently authored tasks by isolating them on physically separate processor nodes. Consequently, it fails to support the principle of modularity.

Digital Equipment Corporation produced an interesting operating system that uses impermeable hierarchical partitioning to provide three distinct sub-operating systems with differing deadline sensitivities, task abstractions and schedulers [VRT96]. In the absence of a name, I dubbed this RTOS "MediaVMS".[16] MediaVMS's hard realtime partition has distributed scheduling, declarative admission control and the performer task abstraction. A planning scheduler similar to Spring's constructs the fully-enumerated scheduling function if the admission control mechanism verifies that the OS can successfully interleave the execution of tasks at their desired invocation rates. As with Spring, MediaVMS relies on declared performer running times and so fails to satisfy the principle of modularity.

SGI's REACT realtime extension to IRIX also uses a distributed scheduler for realtime performer-style tasks [CT00]. REACT uses distributed scheduling for maximum versatility: applications may implement any desired scheduling policy on top of REACT.[17] However, REACT has no admission control mechanism. Consequently, only privileged applications may execute in realtime and so REACT fails to satisfy the principle of modularity.

With the combination of hierarchical partitioning, distributed scheduling, time-triggered externalization and the performer task abstraction, these systems satisfy the principles of processor partitioning, synchronous realtime and ultra-fine granularity realtime. However, all of these systems have only declarative admission control and so cannot schedule tasks successfully

[15] Note that this Spring operating system is completely unrelated to Sun's distributed non-realtime Spring kernel [MGH+95].

[16] The literature did not specify if VMS code was involved but it seems a fitting name.

[17] See [KSSR96] for an example.

unless applications provide correct WCETs. An OS cannot rely on an independently-authored application to do so and so none of these systems can satisfy the principle of modularity.

3.5 Hierarchical Partitioning Operating Systems

Section 3.2 discusses RTOSs that use separate physical processors to guarantee resource allocations to different classes of tasks. Hardware partitioning is inflexible and requires multiprocessor hardware. Hierarchical partitioning provides one possible alternative: the operating system executes different classes of tasks on impermeably partitioned virtual processors.

A hierarchically partitioned operating system is really two or more operating systems in one: a simple realtime "parent" operating system and one or more "child" operating systems nested inside the parent operating system's tasks. Parent operating systems are (sometimes simplified) RMS RTOSs. Implementers choose child operating systems suitable to the application domain.

Many systems use hierarchical partitioning on an RMS parent OS to provide multiple scheduling domains. For example, [KSSR96, AB99, AB98a, GGV96, AB98b, TH97, JLS99, CS01, SCG+00] all present RMS RTOSs enhanced with one or more tasks containing some kind of child operating system. In each case, the parent RMS RTOS provides hard realtime capabilities while children RTOSs executing soft realtime tasks provide improved system utilization and versatility over a pure RMS RTOS.

Yodaiken's RTLinux and Digital's MediaVMS also use hierarchical partitioning to combine tasks with radically different scheduling domains in a single operating system. These two systems are similar to LiquiMedia in their use of hierarchical partitioning and warrant additional discussion.

RTLinux adds hard realtime capability to Linux by virtualizing the hardware's interrupt controllers and running an essentially unchanged Linux kernel on top of a small hard realtime executive [Yod96, Bar97, www00b]. The RTLinux executive layer traps all incoming interrupts and dispatches them to one of its two tasks. The higher priority parent task executes performer-style tasks in response to the incoming interrupts. The second task virtualizes the PC hardware to run a nearly unmodified Linux kernel as its child operating system. RTLinux's use of hierarchical partitioning enabled Barabanov to easily add realtime capabilities to Linux.

To prevent priority inheritance and leave the Linux kernel non-reentrant, the RTLinux executive and non-realtime Linux applications communicate via non-blocking FIFOs that implement asynchronous message passing IPC.

RTLinux realtime tasks run in kernel context and can easily damage the system. RTLinux does not provide an admission control mechanism. Consequently, without either memory

access or processor use protection, RTLinux permits only privileged users to install realtime tasks and so does not satisfy the principle of modularity.

As later advocated by Pawan et al. [PJG+01], Digital's MediaVMS uses an RMS parent operating system to provide an impermeable partition between three child operating systems [VRT96]. Each child operating system provides distinct scheduling and task design aspects: hard realtime performers with distributed scheduling, soft realtime non-preemptive threads with distributed (round-robbin) scheduling and non-realtime time-shared preemptive threads. Oddly, despite including *every* task abstraction, MediaVMS omits an IPC mechanism between tasks in the different child operating systems. While this insures an impermeable partition, it also prevents application developers from appropriately dividing their applications across MediaVMS's versatile combination of scheduling and task abstractions.

As with Spring and REACT, MediaVMS and RTLinux satisfy the the principles of processor partitioning, synchronous realtime and ultra-fine granularity realtime. However, all of these systems provide only declarative admission control and so fail to satisfy the principle of modularity.

3.6 Task Partitioning Operating Systems

The previous section discussed operating systems that use hierarchical partitioning to provide a range of specialized task abstractions and scheduling mechanisms. Many operating systems demonstrate a contrasting design choice: a single task abstraction and scheduling mechanism that attempts to handle all application needs ranging from month-long batch jobs to millisecond response control systems.

Task partitioned operating systems have preemptive threading, rate-based scheduling, permeable task partitioning, divisible task IPC, declarative admission control and soft realtime deadline sensitivity. Figure 3.3 summarizes the location of task partitioning operating systems in the design space.

Given that the task partitioning operating system is a full-blown self-hosted operating system, it represents a popular development project. Relevant examples include the seminal work on Realtime Mach [Raj91]; and Rialto [JIF+96], YARTOS [JB95], DiRT [SAWJ+96], SMART [Nie99] and Scout [BPM99] because the primary intended use of their realtime facilities is the implementation of multimedia applications. Each of these operating systems is discussed in turn below. Other similar examples include ARC-H [Yau99], which combines rate-based scheduling with hierarchical partitioning, and BEST [BB02], which improves Linux's ability to schedule soft realtime tasks by automatically placing them in a rate-based realtime scheduling class.

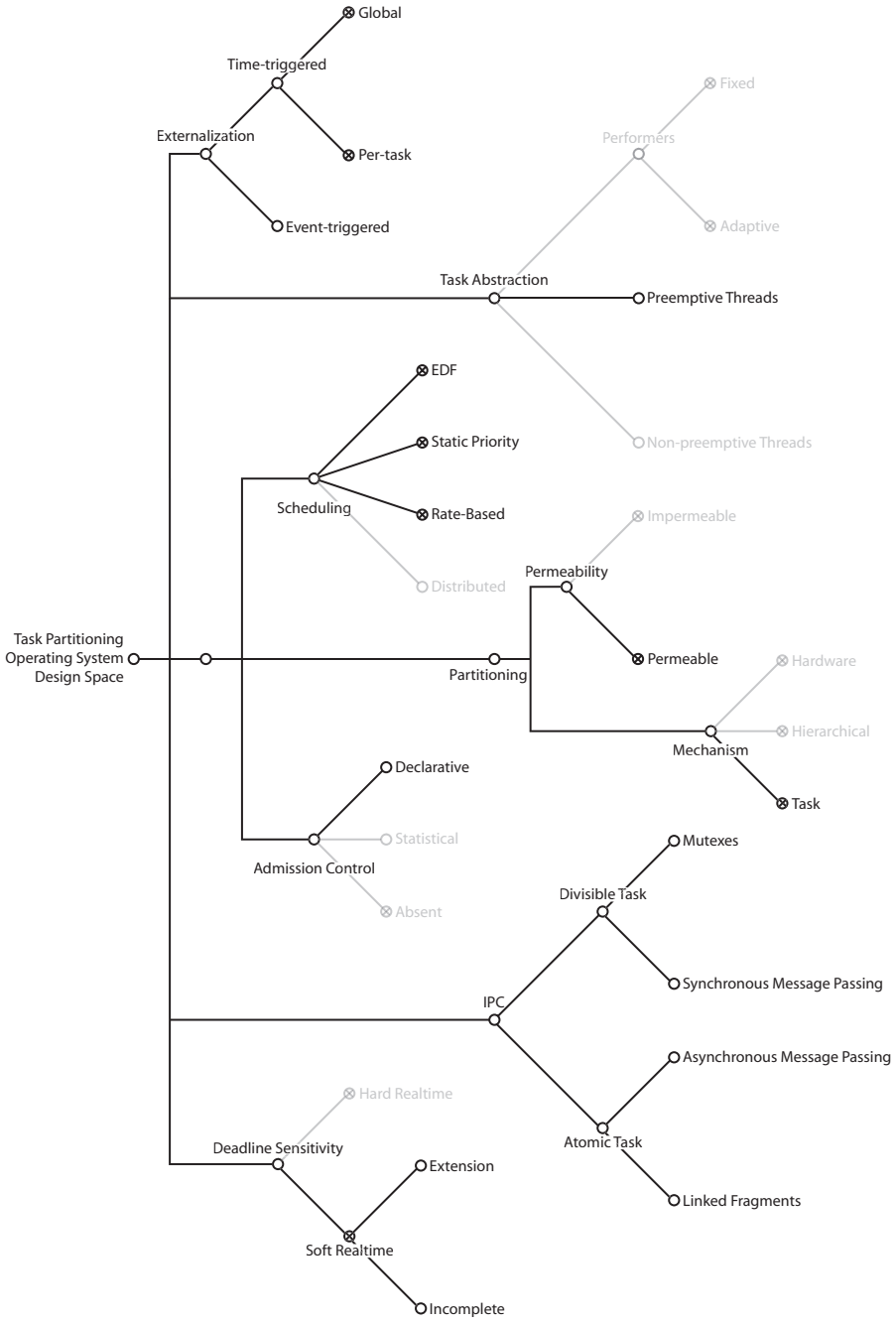

Figure 3.3: Design aspects shared by the task partitioning OS family. Aspects not present in the family are grayed-out.

3.6.1 Realtime Mach and Descendants

Realtime Mach (RTMach) is one of the earliest task partitioned operating systems. Rajkumar created RTMach by adding a priority inheritance mechanism to the basic Mach kernel [Raj91]. Mach is a non-realtime microkernel built around three principal features: preemptive threads, synchronous message passing and flexible memory management [RJO+89]. With the addition of priority inheritance, RTMach supports soft realtime execution with static priority or EDF scheduling, permeable task partitioning and declarative admission control.

In search of superior realtime stream-generation performance, Mercer and Rajkumar extended Realtime Mach with a mechanism called *processor capacity reserves* [MST93, MST94]. A capacity reserve is a reservation of processor time based on a percentage of the processor and a periodicity. An application running under Realtime Mach with capacity reserves (RTMcr) specifies the percentage of processor that it requires and the periodicity of its needs. Consequently, RTMcr augments Realtime Mach with time-triggered externalization, and declarative lifetime admission control.

Mach's micro-kernel architecture significantly complicates RTMcr's implementation of task partitioning. Capacity reserves must allocate fixed proportions of the processor resources per *application* rather than per task. In a micro-kernel, an application task interacts with system services such as a file system by sending a synchronous message to the service task. Consequently, an application using only a single task may cause a significant number of other tasks to consume processor resources. Consequently, RTMcr must correctly "bill" the processor use of system service tasks back to the application tasks which were responsible for invoking them. Tracking these dependencies complicates RTMcr's implementation of task partitioning and increases the implementation overhead of a synchronous IPC operation.

RTMcr provides a choice of either EDF scheduling or static priority scheduling. The static priority scheduler provides a firm task partition but at the cost of reduced utilization. Conversely, RTMcr's EDF scheduler provides higher utilization levels but cannot prevent a habitually overtime application from starving even higher priority applications.

In subsequent research, Rajkumar et al. have added capacity reserves to both Linux [OR98] and Java [dNR00]. The introduction of hierarchical capacity reserves in [MR01] resolves the difficulty of "billing" resource use to specific applications in uncooperative environments and helps satisfy the principle of modularity. However, complexity of usage tracking in these systems precludes satisfying the principle of ultra-fine granularity.

The combination of capacity reserves, preemptive tasks and static priority scheduling satisfy the principles of partitioning and synchronous execution and at least partially satisfy the principle of modularity. However, the cost of tracking an application's execution time across multiple service threads prevents RTMcr and its descendants from satisfying the principle of ultra-fine granularity.

3.6.2 Rialto

Jones et al. designed Rialto to support the execution of independently authored realtime and non-realtime fragments. They intended Rialto to support both traditional time-shared processing tasks and tasks for realtime stream generation [JIF+96, JLDI95, Jon93]. Rialto combines preemptive threads, almost impermeable task partitioning, fluid-flow style rate based scheduling, instantaneous and lifetime declarative admission control, divisible task IPC and soft realtime deadline sensitivity.

Rialto's *resource planner* provides lifetime admission control. An application actually obtains a guaranteed task partition by negotiating with the resource planner. Rialto improves system usability by letting users control inter-application resource negotiation.

Rialto uses fluid-flow scheduling. The scheduler runs threads in order of descending *laxity*. Laxity is a metric ordering threads by deadline and size of *constraint block*. A constraint block is a portion of a thread with realtime constraints and will be discussed further below. Laxity is similar to the lag metric found in DiRT's Earliest Eligible Virtual Deadline First (EEVDF) scheduling. Threads with the smallest estimated constraint block and earliest deadline execute first.

In Rialto, developers specify an application's realtime constraints with a novel mechanism: the *constraint block*. Unlike tasks in other task partitioning RTOSs, Rialto tasks are *not* realtime by default — only their constraint blocks execute in realtime. Developers specify the realtime portions of a task by wrapping them with `BeginConstraint()` and `EndConstraint()` functions. The arguments of `BeginConstraint()` specify the realtime constraints of the enclosed code: its release time, deadline, estimated running time and importance.

Rialto uses these parameters to provide declarative instantaneous admission control: `BeginConstraint()` returns true if the processor can successfully execute all live constraint blocks before their deadlines and after their start times.[18] An application specifies the relative importance of its constraint blocks with the importance argument. The scheduler sacrifices the execution of less important constraint blocks in a transient overload situation.

When combined with task partitioning, constraint blocks partially satisfy the principle of partitioning. They permit developers to partition the expression of realtime and non-realtime code. However, Rialto cannot consistently enforce this partition because it can only partition realtime and non-realtime code along task boundaries and a single task can contain both types of code.

Every new constraint block encountered can change the task's laxity and result in a context switch. Consequently, Rialto imposes a minimum 10ms quantum on task preemptions to improve system efficiency by reducing the number of context switches. The large 10ms quantum fails to satisfy the principle of ultra-fine granularity realtime.

[18] The start time is optional. If no start time is provided, the current time serves as the default.

Rialto provides mutex style IPC. To prevent the consequent vulnerability to priority inversions, it provides *constraint inheritance*. Constraint inheritance generalizes priority inheritance to constraint blocks: blocks in lower laxity tasks inherit the deadlines of blocks in higher laxity tasks that are blocked on them. Constraint inheritance increases the variability of constraint block running time so allows Rialto to provide only soft realtime scheduling.

While only having declarative admission control, Rialto's combination of RBE scheduling and preemptive threads satisfy the principle of modularity. Further, constraint blocks adequately satisfy the principles of partitioning and synchronous realtime. However, Rialto's 10ms thread quantum fails to satisfy the principle of ultra-fine granularity's realtime.

3.6.3 YARTOS and DiRT

Jeffay and Bennett's YARTOS was one of the first task-partitioned operating systems that used rate-based scheduling [JB95]. It has RBE scheduling, preemptive threads, almost impermeable task partitioning, divisible task IPC, declarative admission control and soft realtime deadline sensitivity. YARTOS showed that rate-based scheduling could successfully provide task partitioning and soft realtime task execution.

Subsequently, Stoica et al. extended Realtime Mach with a fluid-flow rate-based scheduler called "proportional share" to implement DiRT [SAWJ+96]. Proportional share scheduling provides a reasonably impermeable task partition based on task importance and a declarative admission control mechanism. DiRT's proportional share scheduling requires time-triggered externalization.

Stoica et al. showed that the DiRT scheduler provides provably correct processor partitioning with an execution latency not more than the smallest possible task period $d'_{p,i}$. Furthermore, DiRT's EEVDF intrinsically handles priority inversion problems by being an extension of EDF scheduling.[19] Thus, the EEVDF scheduler provides impermeable task partitioning. As a result, DiRT does a good job of handling transient overtimes and even tasks that declare incorrect processor requirements.

However, DiRT's EEVDF scheduling, like all rate-based schedulers, has two limitations. First, it cannot provide hard realtime guarantees. Second, EEVDF scheduling is complex and time consuming: it requires $O(\log n)$ time to choose the next executable task from n tasks — a cost incurred at every context switch.

Despite only having declarative admission control, both YARTOS and DiRT satisfy the principle of modularity with a combination of RBE scheduling and preemptive threading. Their task-based partition also satisfies the principle of partitioning while time-triggered externalization and realtime execution satisfy the principle of synchronous realtime. However,

[19] Section 2.5.4 describes the EEVDF scheduling function.

YARTOS and DiRT cannot satisfy the principle of ultra-fine granularity realtime because they both have large context switch and scheduling overhead.

3.6.4 SMART

Nieh implemented SMART by extending Solaris with an RBE-style rate-based scheduler [NL97, Nie99]. Nieh extensively evaluated SMART using real-world loads and optimized SMART for real-world job mixes. SMART demonstrated that, despite its complexity, an RBE scheduler can operate well in practice for a mixture of soft realtime, interactive and non-realtime tasks. However, SMART has relatively poor jitter and deadline accuracy. Consequently, SMART cannot successfully execute hard realtime tasks.

SMART satisfies the principles of partitioning, synchronous realtime and modularity. However, SMART uses Solaris threads and as will be shown in Section 6.4.2, Solaris threads are not sufficiently efficient to support the principle of ultra-fine granularity.

3.6.5 BERT

Bavier et al. implemented a scheduler called "best effort realtime" (BERT) for the Scout operating system. BERT uses fluid-flow rate-based scheduling, augmented with an importance mechanism called "resource stealing".

In BERT, instantaneous declarative admission control prevents transient overloads by having each slice declare its deadline and execution time size. Bavier et al. postulate a mechanism to track the trustworthiness of each task's estimates of its per-slice running time as BERT cannot successfully maintain task partitions in the presence of chronic underestimates of a slice's running time. If a slice over-estimates, other slices can resource steal from its unused share.

Scout also has an lifetime declarative admission control mechanism. The Scout admission control mechanism only accepts tasks if their declared resource and periodicity requirements can be met on average.

3.6.6 Summary

All of the above operating systems occupy a similar location in the design space. They use preemptive threading, rate-based scheduling, divisible task IPC, declarative admission control and nearly impermeable task partitioning. This common set of design space choices is mutually dependent. Task partitioning requires preemptive tasks to successfully partition. Given a successful task partition, an admission control failure is not catastrophic so declarative admission

control is sufficient. Finally, these operating systems favour time-triggered externalization for the same reason as the media generation frameworks: multimedia stream generation is the dominant realtime application of a general purpose operating system and time-triggered externalization best addresses the synchronization needs of multimedia applications.

3.7 Statistical Admission Control

All the operating systems discussed above have either no admission control mechanism or declarative admission control. This section discusses operating systems in the two sub-aspects of statistical admission: declared statistical admission control (DSAC) and empirical statistical admission control (ESAC).

3.7.1 DSAC OSs

All DSAC OSs: [AB99, AB98a, AB98b, TDS+95, ZHS99, MEP01, APLW02] use statistical admission control to increase processor utilization for soft realtime tasks. These systems have soft realtime scheduling, preemptive threads, time-triggered externalization, typically hierarchical partitioning and declarative statistical admission control.

Tia et al. first presented the idea that relaxing WCET constraints on time-triggered task slices improves processor utilization [TDS+95]. Instead of WCET bounds, they used a probability distribution function known to correspond to the distribution of each slice's $t_{p,i,j}$. They showed that probabilistic bounds provide significantly greater processor utilization.

Tia et al. describe two separate operating system architectures using statistical admission control. The first architecture uses hierarchical partitioning with a sporadic bandwidth server executing thread-style tasks inside a single task of an RMS parent operating system. The second architecture combines task partitioning, rate-based scheduling and statistical admission control. Here, Tia et al. extend an EDF scheduler with "slack stealing" to create a fluid-flow scheduler that uses computed estimates of a slice's likely running time for instantaneous admission control.

Atlas and Bestavros simulated a system called SRMS [AB98b]. SRMS extends a traditional RMS preemptive RTOS with statistical admission control. An SRMS task p has a fixed periodicity and a probability density function $\phi_p(t)$ for the processing time of a slice of p. SRMS defines QoS to be the minimum probability that over many releases, the overall task meets its relative deadline d'_p:

$$\text{QoS} = \int_0^{d'_p} \phi_p(t)dt.$$

SRMS admits a set of tasks to the schedule if each task can receive an appropriate average execution allowance, in other words, if QoS exceeds an appropriate minimum value. The

underlying RMS scheduler provides a binary partition between higher priority hard realtime tasks and statistically-scheduled tasks.

SRMS has two potential limitations. First, the SRMS system requires a priori knowledge of $\phi_p(t)$. Second, it can require the performance of statistical calculations in the critical path of the scheduler.

Manolache et al.'s work is a natural extension of SRMS [MEP01]. It implements DSAC for event-triggered tasks with arbitrary precedence relationships. However, it still requires each task to provide the PDF for its execution time.

Zhou et al. simulate another extension of RMS with a probabilistic task model [ZHS99]. They use the task model of Atlas and Tia: time-triggered tasks with a known periodicity and independent running time PDFs. They generalize Liu and Layland's feasibility to tasks with a non-unity probability of meeting their deadlines.

When the running times of tasks are probabilistically bounded, the entire system's feasibility,

$$\text{feasibility} = \prod_{\forall p} \int_0^{d'_p} \phi_p(t)dt, \tag{3.1}$$

is stochastic. Because RMS tasks have different periodicities, a straight-forward evaluation of equation 3.1 does not provide good results. Consequently, Zhou et al. offer a superior mechanism to approximate the system's feasibility using *state tables*, a dynamic programming simulation of a Markov chain.

Abeni and Buttazzo's HARTIK RTOS uses RMS static priority scheduling to partition hard realtime tasks from soft realtime tasks [AB99, AB98a]. All soft realtime task slices run inside a single server thread whose priority is lower than that of any hard realtime (RMS scheduled) task. The soft realtime server provides RBE scheduling.

HARTIK uses lifetime statistical admission control to provide long-running soft realtime tasks with probabilistic QoS guarantees. Soft realtime tasks can be either time-triggered or event-triggered. Event-triggered tasks must provide a probability distribution function for their release times and their WCETs. Time-triggered tasks must provide a PDF for their execution time and a fixed period for their slices. The HARTIK admission control system uses the provided PDFs to confirm that, to a specified probability, the utilization of the system including all hard and firm realtime jobs, is less than 1. As a result, over an arbitrarily long execution window, a task receives a guaranteed average fraction of the processor.

All of these DSAC OSs exchange hard realtime guarantees for improved processor utilization. This trade-off is beneficial for a multimedia operating system where hard realtime execution is unnecessary and can help satisfy the principle of ultra-fine granularity realtime. Moreover, their provision of time-triggered preemptive threads satisfies the principle of processor synchronization. However, as with OSs featuring declarative admission control, all of

these operating systems depend on applications to provide a correct PDF for a task's running time. Trusting an application-provided PDF is arguably even riskier than trusting an application-provided WCET estimate so these OSs fail to satisfy the principle of modularity.

3.7.2 ESAC OSs

Unlike DSAC, ESAC tests tasks without relying on application-provided information. Instead, it admits a task using an estimate of its running time inferred from its past running times. This use of past running times to predict future running times is a form of feedback. Of the ESAC results in the literature, [SLS99, SGG+99, GSPW98, AASD01, ADS96, AAS97, LADS00, ALW99, SSS+02, APLW02] discuss the use of feedback for admission control while [SGG+99] describes an OS that uses ESAC combined with soft realtime, preemptive threads, time-triggered externalization, rate-based scheduling and task partitioning.

In [SLS99], Stankovic proposes that the use of previously observed task running times can improve admission control decisions. He observes that systems such as his own Spring [SR91] and RMS [LL73] require static predetermined knowledge of every task's WCET. Stankovic observes that many embedded realtime systems support control applications. Consequently, he proposes extending a FIFO scheduler with dynamic QoS adjustment driven by high/low watermark levels in the FIFO. Stankovic uses results from classical control theory to ensure a stable and well-behaved scheduler.

Atkins et al. present a more sophisticated application-specific use of observed task performance in [AASD01, ADS96, AAS97, LADS00]. Their CIRCA framework for autonomous flight controllers combines an instrumented RTOS with a realtime middle-ware layer and an AI piloting agent. The piloting agent generates plans to automatically pilot an aircraft without human intervention. These plans correspond to various tasks that the realtime flight controller must execute. The middle-ware layer connects the AI planner with the RTOS via feedback: it delivers accumulated task performance data to the planner. The planner uses this information to constrain its plans to groups of only schedulable tasks.

Alexander et al. present a load balancing scheme for realtime tasks that uses feedback to equalize the loading on multiple processors [ALW99]. Their approach maintains an average of a task's running time in each invocation and uses that estimate to optimize the distribution of the currently running tasks across different (homogeneous) processors.

Sahoo et al. describe and simulate a scheduling algorithm that combines the profiled execution of performer-style tasks with feedback to implement instantaneous ESAC and so satisfies the principle of modularity [SSS+02]. However, their approach has a limitation that precludes satisfying either the principle of ultra-fine granularity or synchronous realtime. Sahoo et al.'s algorithm executes tasks in ascending order by estimated running time and so requires sorting the set of live tasks on every release. This overhead is not compatible with the principle of

ultra-fine granularity. Further, its continual re-ordering of the task invocation order cannot provide the predictable realtime execution required by the principle of synchronous realtime.

Finally, Steere et al. describe an operating system whose fluid-flow rate-based scheduler uses feedback for admission control [SGG+99]. Here, long running tasks receive a variable share of the processor based on maintaining an average progress. Their scheduler records summary statistics for a task's resource use at a particular progress setting. Then, it admits tasks for execution based on a prediction of the task's resource needs at the currently desirable level of progress. Steere et al.'s operating system supports firm, soft and non-realtime tasks.[20] Firm realtime tasks have a fixed progress, the progress of soft-realtime tasks may vary over a task-specified range and non-realtime tasks can have essentially any progress. In [GSPW98], Goel presents a simulation toolkit used by the same research team to explore feedback issues in their OS scheduler.

Of these results, only Steere et al. describe an actual operating system. Its combination of time-triggered tasks, preemptive threading, ESAC, rate-based scheduling and task partitioning can satisfy the principle of processor partitioning, synchronous realtime and modularity. However, as with the task-partitioned OSs discussed in Section 3.6, the minimum quantum size needed to make its rate-based scheduling sufficiently efficient fails to satisfy the principle of ultra-fine granularity realtime.

3.8 Summary

None of results discussed in this chapter simultaneously satisfy the four principles of a multimedia operating system. Obviously the deeply embedded OSs fail to satisfy the principles because they are designed for significantly different application domains such as flight and industrial process control. MGFs, while designed to to support the development of multimedia applications, do not satisfy the principles either because they do not execute these applications in realtime. More general purpose operating systems for multimedia applications fail to satisfy at least one of the four principles. In fact, the general purpose OSs described above can be divided into two groups depending on which three of the four principles that they satisfy.

The first group contains "stout" operating systems: robust general purpose operating systems like Rialto and SMART, which satisfy the principles of processor partitioning, synchronous realtime and modularity, but whose combination of rate-based scheduling and preemptive threading cannot provide the task-invocation efficiency needed to satisfy the principle of ultra-fine granularity.

The second group contains "elitist" operating systems: systems like MediaVMS, REACT, Spring and RTLinux, which satisfy the principles of processor partitioning, synchronous real-

[20] As discussed in Section 2.3, "firm" can still only support soft realtime deadline sensitivity.

time execution and ultra-fine granularity realtime, but do not extend the privilege of realtime scheduling to independently-authored tasks because their scheduling functions and admission control mechanisms depend on tasks truthfully declaring their running times.

Given that both stout and elitist groups satisfy three of the four principles, either could serve as a starting point in the implementation of a multimedia OS. The context switch costs of preemptive threads limit a "stout" OS's ability to satisfy the principle of ultra-fine granularity. Consequently, as will be discussed in Chapter 4, LiquiMedia takes the alternative path: integrate an empirical statistical admission control framework into an "elitist" OS.

Chapter 4

Architecture Overview

LiquiMedia is an operating system designed to schedule multimedia applications. A multimedia application generates realtime information streams for human beings. Consequently, a multimedia operating system should schedule its applications so that they can satisfy constraints imposed by human perception. LiquiMedia is specialized to schedule its applications in this way. Moreover, this specialization takes advantage of limitations in human perception to efficiently support the execution of independently-authored realtime tasks.

Chapter 1 established four principles for the design of a multimedia operating system. The principle of processor partitioning requires explicit separation between realtime and non-realtime tasks both in the operating system and in the mind of the developer. The principle of synchronous realtime requires the OS to schedule tasks so that they can successfully generate information streams by executing synchronized to an external realtime clock from their outset. The principle of modularity requires the operating system to safely execute multiple independently authored realtime and non-realtime tasks. Finally, the principle of ultra-fine granularity realtime requires the operating system to efficiently interleave the operation of a large number of tiny realtime fragments.

None of the operating systems discussed in Chapter 3 satisfy all four principles. LiquiMedia satisfies them with a unique combination of the design aspects described in Chapter 2 — making LiquiMedia unique in both approach and capabilities. The remainder of this chapter describes the design aspects comprising LiquiMedia and discusses how they cooperate to satisfy the principles. Appendix B extends this description to a level of detail suitable for implementers.

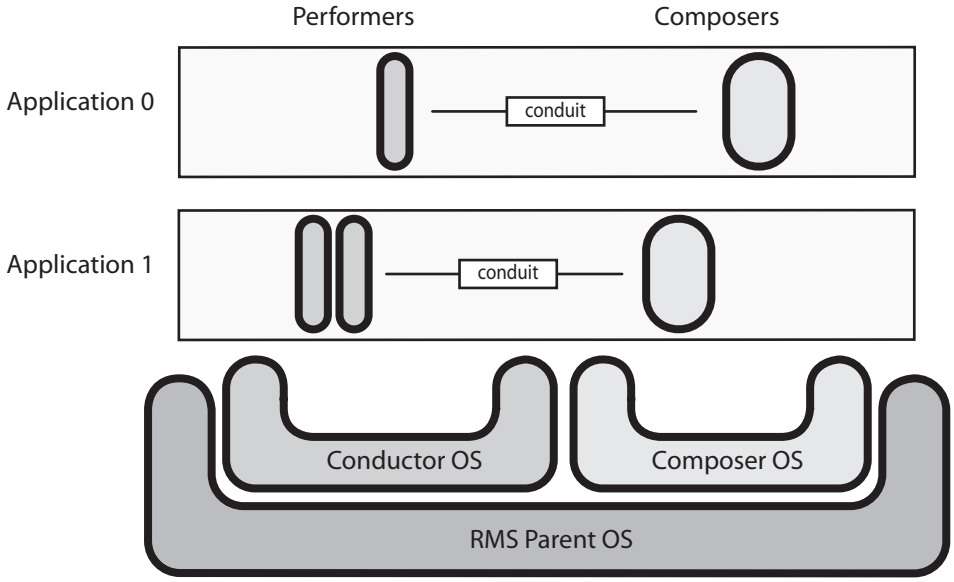

Figure 4.1: Each application consists of one or more composers and one or more performers. These two different kinds of tasks execute in separate sub-operating systems and communicate via conduits.

4.1 Taxonomic Position

Chapter 1 introduced the recital model as a generic architecture for multimedia applications. Consequently, every LiquiMedia application is an instance of the recital model: realtime and non-realtime tasks communicating over asynchronous channels. Drawing on the recital model for nomenclature, LiquiMedia's realtime tasks are called performers and its non-realtime tasks composers.

Performers and composers are so different that it is impossible to support both with a monolithic operating system's single choice of task abstraction and scheduler. Instead, LiquiMedia uses hierarchical partitioning to host two separate sub-operating systems inside a parent operating system. The realtime conductor sub-operating system executes performers while the non-realtime sub-operating system executes composers. Each sub-operating system is therefore specialized for its respective client tasks while the parent operating system is specialized to run these two sub-operating systems.

As shown in Figure 4.1, each LiquiMedia application consists of both composer and performer tasks. Furthermore, LiquiMedia can execute more than one application. All the performers from all applications run inside the single conductor sub-operating system and similarly all composers from all applications run inside the single composer sub-operating system.

Because of its use of hierarchical partitioning, LiquiMedia contains three different operating

systems with different design aspects:

- the parent OS,
- the conductor OS, and
- the composer OS.

Each of these operating systems has different design aspects. Figure 4.2 shows the design aspects comprising LiquiMedia's three sub-operating systems and hence LiquiMedia's position in the taxonomy defined in Chapter 2.

LiquiMedia's parent operating system combines RMS scheduling and preemptive threads to provide an impermeable partition between the conductor and composer sub-operating systems. As discussed in Section 2.5.2, an RMS OS can only maintain the partition if it forbids synchronous IPC between its tasks. However, the composers and performers comprising a LiquiMedia application must be able to communicate. Consequently, LiquiMedia provides asynchronous message passing IPC between performers and composers.

Considered in isolation, the conductor has the following design aspects: global time-triggered externalization, soft realtime deadline sensitivity, permeable task partitioning, fixed performer-style tasks, distributed scheduling, atomic task IPC based on asynchronous message passing both internally and to composer tasks and statistical admission control.

The conductor uses the performer task abstraction defined in Section 2.2.1. The principle of ultra-fine granularity requires performers because they are the most efficient task abstraction. LiquiMedia uses time-triggered externalization for its performers because the combination of time-triggered externalization and performers satisfies the principle of synchronous realtime execution.

The principle of ultra-fine granularity realtime encourages the development of applications with large numbers of performers. Only distributed scheduling's $O(1)$ cost of task selection can efficiently support many small realtime tasks. Consequently, LiquiMedia uses distributed scheduling for performers.

LiquiMedia has empirical statistical admission control (ESAC) in both instantaneous and lifetime contexts. LiquiMedia uses instantaneous admission control to avoid transient overload. LiquiMedia uses lifetime admission control to verify that each application's contribution to the schedule is feasible prior to commencing its execution.

LiquiMedia has distributed scheduling. Each application controls its own contribution to the schedule. Distributed scheduling permits separating the production of a schedule from its use for the dispatching of tasks. As a result, LiquiMedia's scheduler executes in a composer. Thus, LiquiMedia can use an expensive lifetime admission control mechanism without violating the principle of ultra-fine granularity realtime.

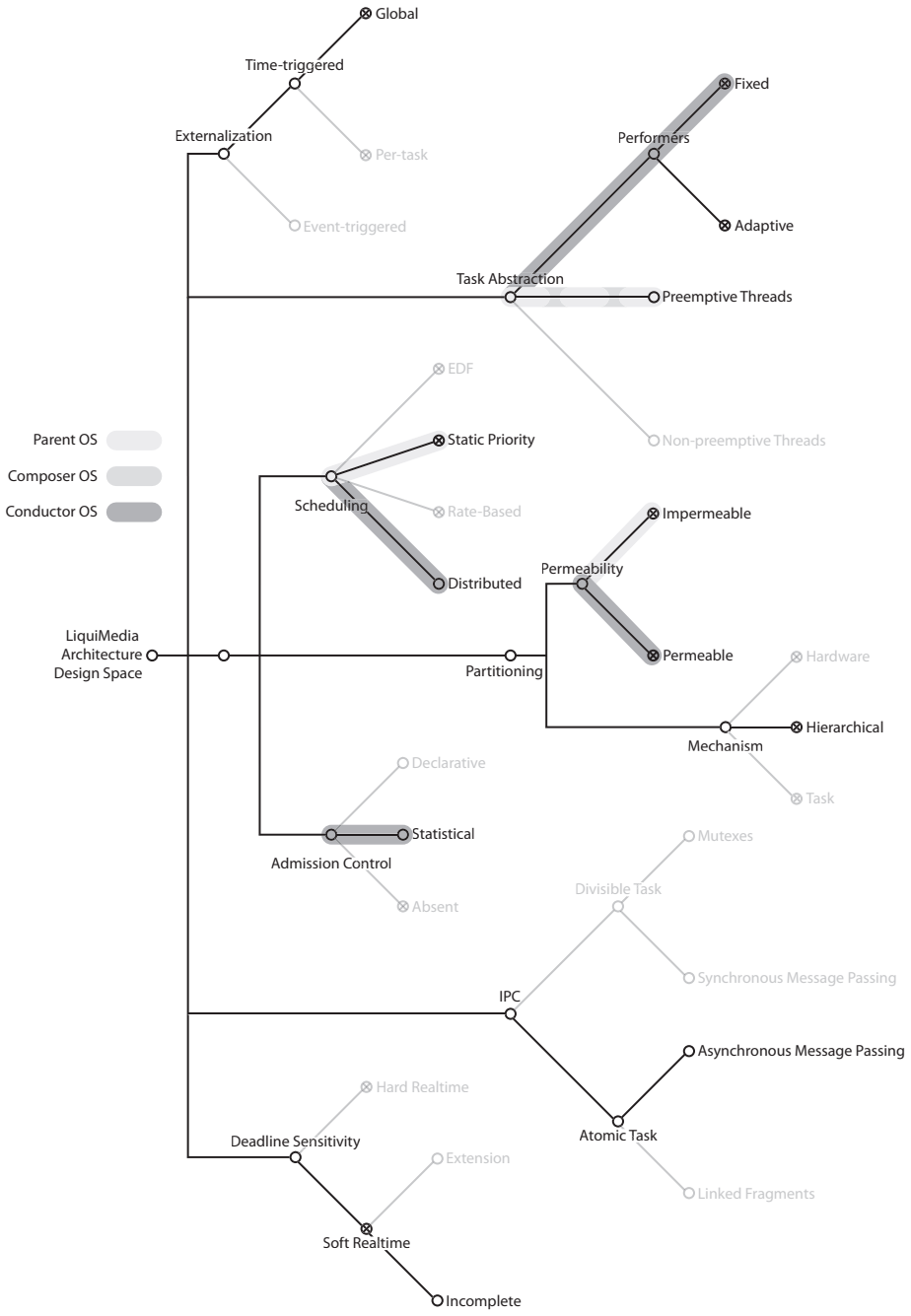

Figure 4.2: LiquiMedia's chosen design aspects from the taxonomy defined in Chapter 2. Aspects not present in LiquiMedia are grayed-out.

The composer sub-operating system is a standard time-shared general purpose operating system. It has the following design aspects: non-realtime operation, time-shared scheduling, preemptive threads, divisible task IPC and (internal) task partitioning.

4.2 Soft Realtime

LiquiMedia offers performers soft realtime scheduling because multimedia applications can tolerate occasional scheduling failures and permitting these failures enables LiquiMedia's support for independently-authored realtime applications.

Multimedia applications can tolerate occasional scheduling failures for three reasons. First, most multimedia applications exist for user entertainment or communication. Occasional scheduling failures in these domains do not have the same life-critical consequences that can occur with applications such as flight controllers. Instead, entertainment applications benefit from implementation approaches that trade hard realtime guarantees for improved processor utilization and consequently lower costs.

Second, at least in the visual modality, human observers have a sizable tolerance for the occasional dropped frame. For example, users regularly watch realtime information streams encoded as PAL, NTSC and film image sequences despite the fact that they respectively display frames at 25, 29.97 and 24 Hz [Poy96]. Given this tolerance, a video playback performer with a nominal target of 30 frames per second remains satisfactory even when scheduling failures drop 5% of the frames.

Third, the architecture of many multimedia applications includes one or more basic periods of buffering. For example, an MPEG player similar to the example in Section 4.8 buffers several video frames. As a result, a scheduling failure in basic period i will cause no detectable interruption in the video stream if all performers execute successfully in basic period $i + 1$.

LiquiMedia takes advantage of soft realtime's tolerance for occasional scheduling failure to schedule realtime tasks based on a probabilistic model (c.f. B.2). Compared to worst case execution time (WCET) scheduling, probabilistic scheduling has three benefits: it quantifies "soft" realtime; it increases utilization and, most importantly, it supports independently authored realtime tasks.

First, unlike the soft realtime operating systems discussed in Chapter 3, LiquiMedia's probabilistic scheduling has a quantitative notion of soft realtime: firmness. In LiquiMedia, the firmness P_c measures the probability that a realtime task completes executing before its deadline. Every task in a hard realtime operating system must have a firmness of 1.0. In contrast, developers or a user can configure LiquiMedia to schedule tasks at any firmness $P_c < 1.0$.

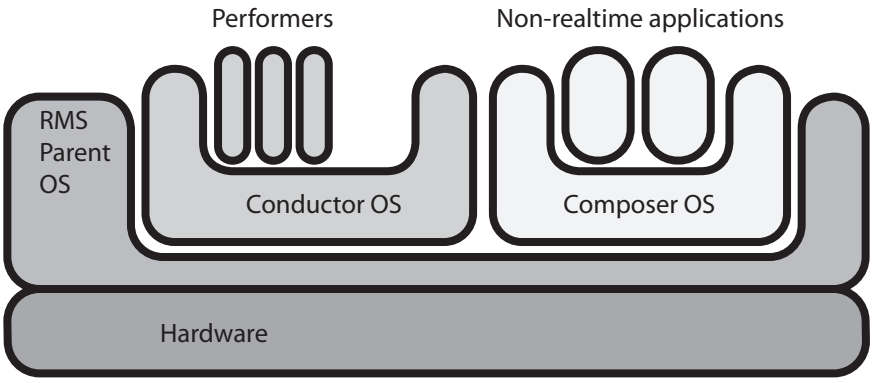

Figure 4.3: A diagrammatic representation of hierarchical partitioning.

Second, probabilistic scheduling can provide higher utilization than WCET scheduling. For example, Chapter 6 shows that a signal-processing performer scheduled probabilistically at a firmness level of 0.99 has a utilization 2.6 times better than when scheduled with WCET. This is a compelling utilization advantage for multimedia applications such as video games.

Third, probabilistic scheduling permits LiquiMedia to support independently authored untrusted realtime tasks. An operating system cannot safely assume that an independently authored task will correctly declare its resource needs: realtime code frequently contains defects that alter its running time [Wir77]. Some realtime code may even deliberately declare false resource needs to impede the operating system's execution of other tasks. Whether through error or malice, a task can declare invalid resource needs. Rather than depending on a task's declaration of its resource needs, LiquiMedia uses statistical inference to predict the running times of realtime tasks from an accumulated record of their past running times. Though allowing only soft realtime deadline sensitivity, using a probabilistic model to predict a task's running time is the only way an operating system can safely schedule independently-authored realtime tasks.

4.3 Hierarchical Partitioning

As discussed above, LiquiMedia uses hierarchical partitioning because it provides separate sub-operating systems whose scheduling and task abstractions can be chosen independently. LiquiMedia has two sub-operating systems: the realtime conductor (c.f. B.3) and non-realtime composer sub-operating system. Figure 4.3 shows the operating system hierarchy schematically.

LiquiMedia uses an RMS RTOS for its parent operating system. When free of priority inversions, an RMS RTOS can provide a completely impermeable partition between its child

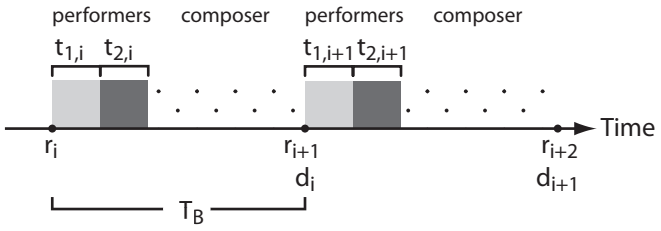

Figure 4.4: The structure of a basic period.

operating systems. LiquiMedia permits only asynchronous message passing between the two sub-operating systems as this eliminates the possibility of a priority inversion and thereby insures an impermeable partition.

LiquiMedia permits only a single conductor sub-operating system. Section 7.3 discusses future generalizations capable of supporting multiple conductors. The composer sub-operating system can freely schedule tasks on multiple physical processors as permitted by the parent RTOS.

The composer operating system has a period of unlimited duration and runs an arbitrary time-shared task mixture indefinitely. The conductor executes as an RMS task, acquiring the processor every T_B seconds when a hardware timer fires and preempts whichever sub-operating system is currently executing. (As discussed in Section B.3.7, the parent RMS OS can invoke the conductor recursively.) Figure 4.4 shows the timing structure that results: the parent OS activates the conductor every T_B seconds and the conductor then invokes every performer. Any time remaining at the end of the conductor's execution is available to composer tasks.

Use of hierarchical partitioning permits an arbitrary choice of composer operating system. The LiquiMedia prototype uses Solaris for its mature system services and implementation convenience. Using a UNIX flavour like Solaris for LiquiMedia's composer operating system results in a versatile system. Modern UNIX operating systems have excellent support for complex non-realtime applications. However, they have no support for independently-authored realtime tasks. Hierarchical partitioning permits the addition of the conductor OS, which removes this limitation without compromising UNIX's existing capabilities.

Practicality demands that interrupt handlers preempt the conductor because LiquiMedia must respond to many external I/O devices in less than the basic period T_B. On the arrival of a hardware interrupt, its service routine suspends the conductor, executes and then returns. The conductor includes the running time of the interrupt service routine in its measurement of the running time of the interrupted performer to simplify the implementation. Section 6.3 discusses some of the difficulties of this approach.

The LiquiMedia architecture supports a mixture of non-realtime and soft realtime tasks. An extension to the architecture could support hard realtime tasks with periods less than

the basic period T_B by adding additional sub-operating systems executing as higher priority preemptive RMS tasks in LiquiMedia's parent operating system.

LiquiMedia's use of hierarchical partitioning permits it to satisfy the principle of processor partitioning. Non-realtime code can never interfere with the execution of realtime code. Further, hierarchical partitioning provides different task and scheduling abstractions for realtime and non-realtime tasks. While this requires developers to learn two different abstractions, they are abstractions for two distinctly different application domains. Non-realtime applications have negligible architectural similarity with realtime applications and the application's programming model should reflect it.

4.4 Realtime Performers

Realtime tasks in LiquiMedia use the performer-style task abstraction (c.f. B.1) for two reasons. First, performers are a practical and natural choice for media processing. Second, only performers permit an implementation sufficiently efficient for ultra-fine granularity realtime.

4.4.1 Practical and Natural

The performer task abstraction is practical because examples and programming expertise exist. This expertise and examples exist because the media generation frameworks (MGF) discussed in Chapter 3 use performer-style tasks. Furthermore, practical use of the LiquiMedia architecture involves porting media processing components from an MGF to LiquiMedia. The MGF components are already written as performers so LiquiMedia's use of the performer-style task abstraction minimizes the porting effort.

The performer task abstraction is a natural choice because its combination of atomic and synchronous execution resembles the mathematical structure of digital signal processing applications [OS94]. Each invocation of a performer processes a single sample in a uniform fixed rate sample stream such as a sequence of video frames or audio samples. Furthermore, every performer remains synchronized with every other performer so applications can can easily compose performers together into filter chains.

4.4.2 Efficient

The performer task abstraction is also efficient because performers voluntarily relinquish the processor at the end of their execution and thereby reduce the overhead of switching between performers to that of a function call. In contrast, threaded task abstractions have the overhead of a full context switch when switching between tasks. Context switching costs for multimedia

tasks are increasing as processors incorporate special multimedia execution units. For example, Intel's MMX multimedia extension adds eight 64-bit registers to the IA32 micro-architecture while SSE adds a further eight 128-bit registers [Int99]. VLIW systems take this trend further.

Combining distributed scheduling with the performer task abstraction permits even greater efficiencies. With distributed scheduling, an application can fix the order of performer execution and hence optimize cache and register usage across performer invocations while still exposing individual performers to system-wide scheduling. In contrast, application developers cannot control the interleaving of preemptive threads and so can neither optimize cache nor register usage across threads.

4.5 Non-realtime Threads

While performers are an efficient and natural task abstraction for realtime media-generating tasks, they are not appropriate for long-running non-realtime tasks. Instead, for long running non-realtime tasks, LiquiMedia uses the preemptive threading task abstraction. The LiquiMedia prototype uses Solaris as its composer operating system. Consequently it inherits all of Solaris's design aspects including preemptive threading [Bac86, GC94]. In the LiquiMedia architecture, traditional non-realtime applications such as compilers, shells and mail clients execute as preemptive threads because, as UNIX's enduring success clearly demonstrates, the preemptive thread abstraction is satisfactory for long-running non-realtime applications.

4.6 Conduit IPC

As described in Sections 4.1 and 4.3 above, a LiquiMedia application consists of both performers and composer tasks. However, hierarchical partitioning rigidly separates these two different types of tasks to prevent composers from interfering with performer execution. LiquiMedia's conduit inter-process communication (IPC) facility permits composers and performers to communicate while preserving this partition.

Conduits are an asymmetric one-directional message passing facility with a design inspired by sockets in UNIX [LMKQ89] and conduits in [NK91]. As shown in Figure 4.5, each conduit has a reader endpoint and a writer endpoint. A task adds messages atomically at the writer endpoint. A task obtains messages atomically at the reader endpoint. A conduit stores messages internally in a ring buffer.

A conduit's asymmetry preserves the partition between the composer and conductor sub-operating systems even while permitting communication between them. From the conductor's view point, a conduit is a non-blocking FIFO and has asynchronous message passing. As a

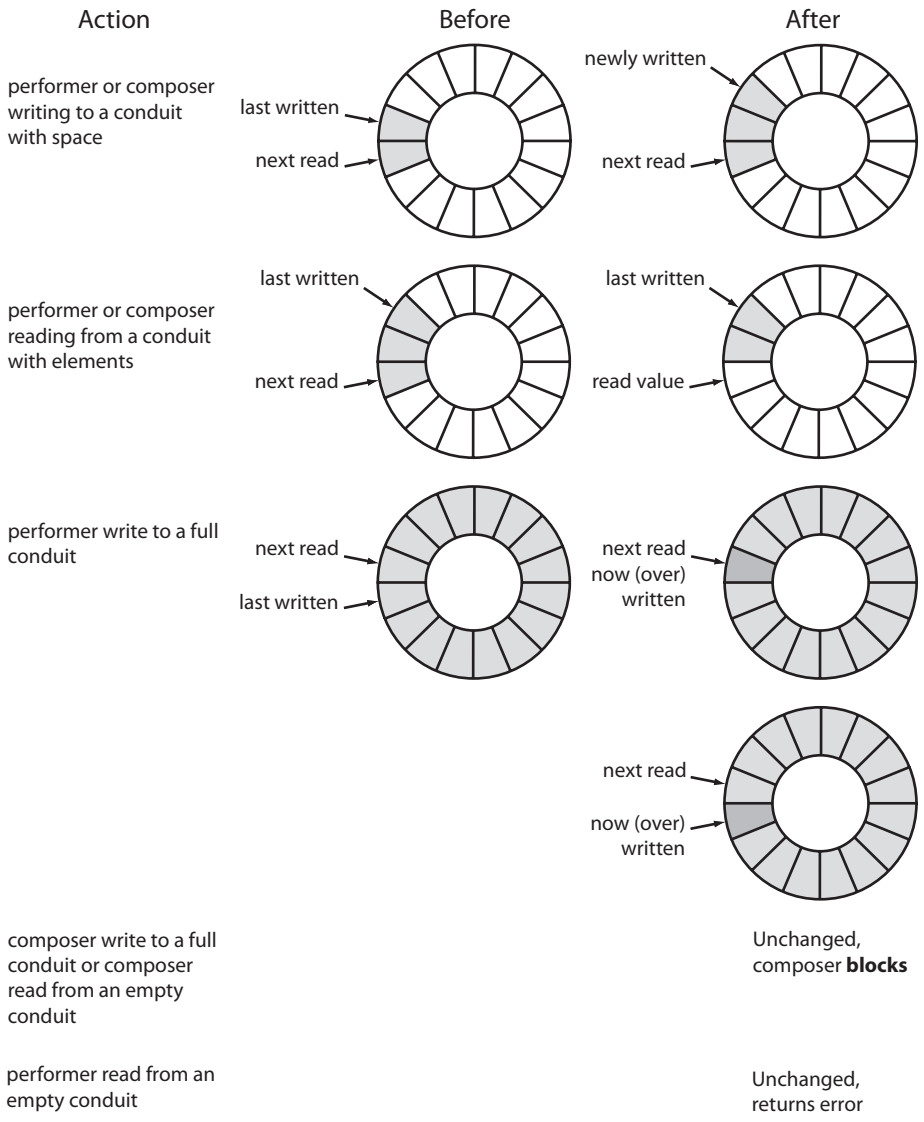

Figure 4.5: The operation of the conduit IPC mechanism.

result, the conductor need never wait for a composer thread to add or remove an item from a conduit. Instead, the conductor continues executing without the risk of a priority inversion imposed by synchronous IPC.

From a composer thread's viewpoint, a conduit has the semantics of a UNIX pipe: a composer writing to a conduit blocks when the ring buffer is full while a composer reading from the conduit blocks when the ring buffer is empty. These semantics are a natural choice for ease of software development given that LiquiMedia's composer OS is a flavour of UNIX.

To make it possible to write library code usable in both performer and composer tasks, conduits can also be used for performer to performer and composer to composer communication. When used for performer to performer communication, both of a conduit's endpoints have have non-blocking FIFO semantics while a conduit acts as a UNIX pipe for composer to composer communication.

Besides permitting communication between sub-operating systems without sacrificing the partition between them, conduits are also a practical IPC mechanism for multimedia applications such as the LiquiMedia audio player. Described further in 6.2, the LiquiMedia audio player combines an audio decompression composer with filter and output performers. A conduit carrying audio samples was the ideal communication mechanism between these tasks because it provided the "best-effort" synchronization needed between a non-realtime (and non-synchronous) composer and synchronous realtime performers.

4.7 Task-partitioned Performers

The conductor uses the instantaneous admission control mechanism discussed in Section 4.9 to provide a permeable task partition between performers. Before executing a performer, the conductor verifies that the basic period contains sufficient remaining time for its execution. If insufficient time remains because previously executed performers ran longer than predicted, the conductor defers the performer's execution.

As a result, every performer must be prepared to address the situation where the conductor has not executed it for one or more previous basic periods. The conductor informs each performer on invocation of the number of preceding basic periods in which the performer has not been invoked. Most MGF applications have an obvious response such as skipping frames or advancing the step size of a simulation.

4.8 Schedule Graph

The conductor uses a generalization of distributed scheduling that I call a *scheduling graph*. A schedule graph represents the schedule as a DAG where a node is a performer and its outbound

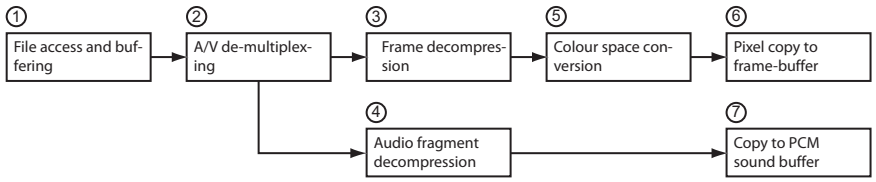

Figure 4.6: A JMF-inspired filter graph for MPEG playback.

edges point to all possible immediately subsequent performers (c.f. B.4.1). At the end of its execution, a performer indicates to the conductor which outbound edge specifies the next performer. Unlike the simple distributed scheduling found in a cyclic executive, the schedule graph can represent the graph structure of tasks found in MGFs without sacrificing the $O(1)$ scheduling function required to satisfy the principle of ultra-fine granularity realtime.

To satisfy the principle of ultra-fine granularity, the conductor requires a scheduling function that it can always evaluate in $O(1)$ time. The simple linear table of tasks found in a cyclic executive provides the necessary performance but is too inflexible to represent the performer structures needed to implement a multimedia application with a MGF.

For example, Figure 4.6 shows the task structure for an MPEG playback application built by the JMF MGF.[1] The tasks in the graph form a data-flow graph: media data flows from left to right and is processed in sequence by each component. Performers, however are time-triggered and so the data-flow graph must be converted to a linear sequence of performer executions of the form: $1, 2, 3, 4, 5, 6, 7$.[2] Simple distributed scheduling can easily represent this sequence.

However simple distributed scheduling has two limitations in this context. First, to satisfy the constraints of statistical scheduling, LiquiMedia requires that all invocations of a performer have similar execution times. The MPEG frame decompressor shown in Figure 4.6 has different execution times depending on whether the subject frame is an I, B, or P style frame [Tek95]. Replacing the MPEG decompressor with I, B and P frame specific performers solves this problem.

Second, the performers should be reordered to take advantage of importance. All performers have an importance value determined by their place in the schedule. As described further in Section 4.10, the later the performer lies in the schedule, the more likely it is to be deferred. Consequently, an application should attempt to arrange its performers in order of decreasing importance.

[1] JMF includes a utility that can display the filter graph constructed by a JMF application [Jav00]. This utility is the source of task graph shown in Figure 4.6.

[2] The conversion could be performed mechanically. Because portions of the data-flow graph occur in parallel, other valid sequences exist.

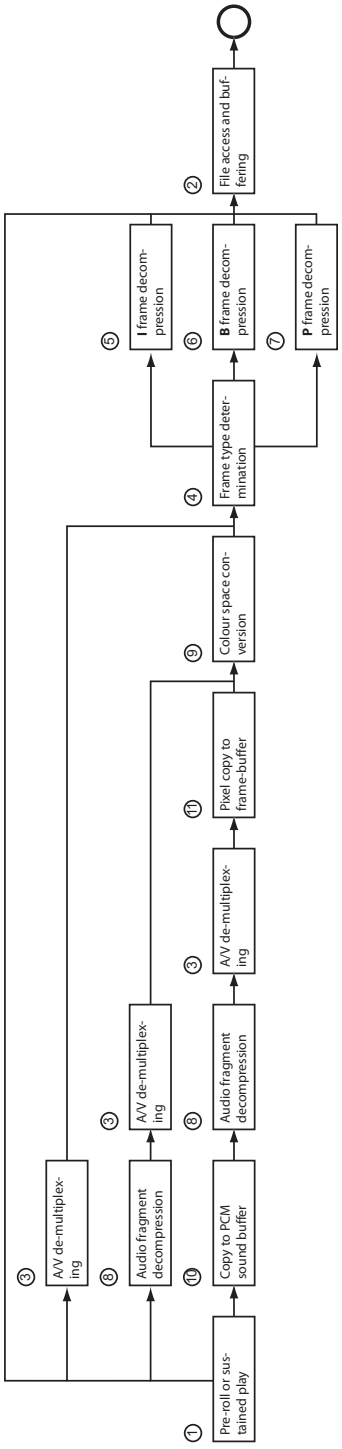

Figure 4.7: The final schedule graph for a LiquiMedia MPEG player.

Figure 4.7 shows the final form of the example. In it, the performers are in order of decreasing importance with audio decompression favoured over video decompression. Also, the graph features the paths necessary to handle both steady-state playback and pre-roll. The directed graph contains multiple paths where each path defines a possible performer execution sequence:[3]

$$\{1, 2\}$$
$$\{1, 3, 4, 5, 2\}$$
$$\{1, 8, 3, 9, 4, 5, 2\}$$
$$\{1, 8, 3, 9, 4, 6, 2\}$$
$$\{1, 8, 3, 9, 4, 7, 2\}$$
$$\{1, 10, 8, 3, 11, 9, 4, 5, 2\}$$
$$\{1, 10, 8, 3, 11, 9, 4, 6, 2\}$$
$$\{1, 10, 8, 3, 11, 9, 4, 7, 2\}.$$

Such a task graph cannot be represented by the cyclic executive's linear list of tasks and instead requires the more general schedule graph representation.

Fortunately, this more general representation does not increase the overhead of the conductor's scheduling function. An adjacency list representation for the schedule permits the conductor to contain an $O(1)$ implementation of the scheduling function [CLR90]. The adjacency-list representation provides both a flexible schedule structure and the efficiency required by the principle of ultra-fine grain realtime.

The above example also demonstrates a general-purpose strategy for reducing performer variance. Developers should sub-divide a large monolithic performer such as performer 3 in Figure 4.6 into the multiple simpler performers $\{4, 5, 6, 7\}$ found in Figure 4.7 whenever the monolithic performer chooses to execute one of several different functions in a basic period. A multi-function performer's variance is the sum of squares difference between its average running time and the running times of the individual component functions. Dividing the multi-function performer into separate performers lowers the total variance and better exposes the application's structure for more advantageous scheduling.

4.9 Statistical Admission Control

LiquiMedia satisfies the principle of modularity with its two ESAC admission control mechanisms. The scheduler uses the *lifetime* admission control mechanism to insure that an entire

[3]The graph permits two additional paths: $\{1, 3, 4, 6, 2\}$ and $\{1, 3, 4, 7, 2\}$. Because MPEG playback must begin on an I frame, these paths are never executed.

schedule graph (the average of many performers) has sufficient firmness while the conductor uses *instantaneous admission control* in the instant before invoking a performer to insure that the performer has a sufficient probability of executing in realtime.

4.9.1 Lifetime Admission Control

The schedule graph provides the foundation for both admission control mechanisms. The conductor executes performers as specified by the *current schedule*. Each performer node in this graph includes a probabilistic estimate of the performer's running time. The conductor uses this estimate for its instantaneous admission control mechanism. Should the conductor admit a performer and the performer execute successfully, the conductor also records the statistical information described in Section B.3.6 in the performer's node in the schedule graph.

The scheduler's lifetime admission control mechanism regularly re-verifies that the current schedule has at least the required minimum probability of realtime execution. During the course of this re-verification, it closes the feedback loop between the conductor's collection of statistical information and its use of estimates of a performer's running time (c.f. B.4.6).

The scheduler also invokes its admission control mechanism in response to an application requesting an alteration of the schedule graph. In this case, the scheduler first constructs a new schedule combining all the unaltered sub-graphs of the current schedule with the application's submission and then admission tests the new schedule. The scheduler combines application sub-graphs in their order of submission — placing the newest sub-graph last. Should the new schedule prove admissible, the scheduler passes it to the conductor and the new schedule becomes the current schedule at the conductor's earliest convenience.

The admission control test has the same four stages regardless of whether the scheduler is applying it to a newly created schedule or re-testing the conductor's current schedule. First, the scheduler computes a probabilistic bound on each performer's running time and attaches it to the subject schedule graph. Second, the scheduler uses these per-performer estimates to compute a probabilistic bound on the running time of each path in the schedule graph. Third, the scheduler tests that the estimated running times of all such paths remain less than the duration of a basic period and, if they do, passes the schedule graph to the conductor. Otherwise, the subject schedule has experienced a *lifetime admission failure* and the scheduler invokes the fourth stage: it removes the least important performers from the schedule and repeats the admission process on the now reduced subject schedule.

In the first stage, the scheduler computes a probabilistic bound on each performer's running time. It estimates a performer's running time with Chebyshev's inequality because Chebyshev's inequality holds regardless of the distribution of the performer's running time [WM78]. Section B.2 presents the mathematical details.

Use of Chebyshev's inequality requires the mean and standard deviation of a performer's running time. As described in Section B.3.6, the scheduler uses Chan et al.'s pair-wise algorithm to compute these summary statistics from the invocation count, sum of performer running times and sum of squared performer running times [CGL83]. The conductor collects these values by measuring a performer's actual running time on every invocation with a hardware cycle counter.

An un-executed performer does not have collected statistical information. Consequently, LiquiMedia requires that applications provide estimates of each performer's standard deviation and mean running time. The admission control mechanism uses these values when admission testing a schedule containing new performers.

In the second stage of the admission control test, the scheduler enumerates the paths through the schedule graph via depth-first graph traversal. For each path, it combines the per-performer estimates as described in Section B.4.5 and estimates the total running time of the path.

In the third stage, the scheduler verifies that all of the previously calculated per-path estimates are less than the basic period. Finally, only if one or more of the path estimates exceed the basic period will the scheduler enter the fourth stage: resolving a lifetime admission failure by removing the last-most performer from each inadmissible path until the schedule becomes admissible.

A lifetime admission failure can occur in three situations. First, an application's initial estimate of its performers' running times can show that these un-executed performers are inadmissible. Second, while the scheduler may have admitted new performers based on an application's initial estimate of their running time, they may not be admissible once the scheduler has computed estimates of their running times from conductor-collected measurements. Third, a long-running performer may suddenly begin to take significantly longer to execute.

A lifetime admission failure can be severe because correcting it can interrupt one or more existing realtime streams. However most such failures are harmless because they are caused by the first two situations described above. In the first situation, the affected performers have never executed and so removing them from the schedule does not interrupt an existing realtime stream.

In the second situation, the scheduler admission tests and removes the affected performers from the schedule before the user perceives them as realtime stream. The lifetime admission control mechanism can test a schedule within a user's 200 millisecond segregation threshold because, as shown in Section 6.5.3, the conductor provides the scheduler with per-performer summary statistics sufficiently accurate for admission testing within 10 basic periods or 130 milliseconds.

Unfortunately, the third situation under which a lifetime admission failure can occur does disrupt established streams. In the third situation, a performer embedded inside the current

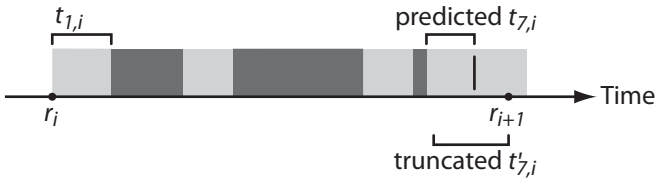

Figure 4.8: An example of an *overtime* scheduling error: performer 7 has not finished executing at the end of the basic period.

schedule increases its running time or variance to such an extent that the schedule becomes inadmissible. For example, assume that performer 9 in Figure 4.7 contains a programming error that causes its execution variance to increase in proportion to the number of displayed frames. Eventually, the error renders the performer inadmissible and the scheduler will remove the sub-path $\{9, 4, 7, 2\}$ and consequently interrupt the video stream.

However, the performers $\{1, 10, 8, 3, 11\}$ continue to operate and so the video's audio stream remains unaffected. This example shows how LiquiMedia provides importance: performers early in the schedule graph are preserved at the expense of later performers. Because a performer's importance is tied to its position in the schedule graph, an application can easily specify the relative importance of its performers with its schedule graph.

The lifetime admission failure is one of three scheduling errors that LiquiMedia can encounter. The other two are overtimes and deferrals. Figure 4.8 shows an example of an overtime: performer $p = 7$ has not completed executing at the beginning of the next basic period. When this occurs, the conductor suspends the execution of performer p, notifies via conduit any interested composer threads and does not attempt to execute p in subsequent basic periods. Overtimes are LiquiMedia's most severe type of scheduling error because they always interrupt the generation of a realtime stream.

4.10 Instantaneous Admission Control

Because overtimes are so serious, the conductor uses its instantaneous admission control mechanism to substitute a deferral for the more serious overtime scheduling error. A deferral occurs when the conductor's instantaneous admission control mechanism detects and avoids a potential overtime. As shown in Section 6.5.2, this mechanism eliminates all overtimes in schedules comprised of standard test performers.

Figure 4.9 shows an example. Here, performer 7 has less than probability P_c of executing before the end of basic period j so the conductor defers its execution.

The conductor implements the instantaneous admission control mechanism (c.f. B.3.5). Before invoking a performer, it tests if the the time remaining in the current basic period

Figure 4.9: An example of a *deferral*. The shaded blocks indicate the running times of performers $1 \ldots 6$ whose sum exceeds $\mathrm{lt}(7, P_c)$ — the latest P_c-likely start time of performer 7.

exceeds the P_c-likely estimate of the performer's running time. Only if the time remaining exceeds the estimate will the conductor invoke the performer. The scheduler computes a performer's estimated running time each time that it re-verifies the schedule.

The conductor counts the number of overtimes taken by any performer. During re-verification, the scheduler suspends performers which have experienced an unacceptable number of overtimes. As with all other circumstances where a performer is suspended, LiquiMedia's scheduler notifies interested composers.

Besides illustrating a deferral, Figure 4.9 also shows why LiquiMedia ties importance to a performer's position in the schedule. The later the conductor executes a performer in the basic period, the more likely it is that the performer experiences a deferral or an overtime scheduling error because with each successive performer executed, the time remaining to execute a performer decreases. As a result, while the admission control mechanisms insure that every executed performer has at least a firmness of P_c, earlier performers have a higher probability of executing successfully. LiquiMedia takes advantage of this property to provide per-performer importance simply by controlling the structure of the schedule graph.

Deferrals are the least serious scheduling error because they only briefly interrupt stream generation. For example, a single dropped video frame passes unnoticed and audio hardware has sufficient buffering to hide a single deferral in a performer generating an audio stream.[4]

The instantaneous admission control mechanism helps LiquiMedia to support higher realtime loadings without damaging the user experience. As shown in Section 6.5.2, the instantaneous admission control mechanism eliminates overtimes at schedule loadings which would otherwise always experience them. By converting overtimes into deferrals, LiquiMedia can execute highly loaded schedules without subjecting users to the unpleasant gaps in a realtime stream caused by performer overtimes.

[4]For example, the SGI VisualWorkstation has an 8kB hardware audio buffer which holds approximately 3.5 basic periods of CD quality audio [Sil99].

4.11 Satisfies Fundamental Principles

LiquiMedia is a multimedia operating system. Chapter 1 defines four principles that a multimedia operating system must satisfy: the principle of processor partitioning, synchronous realtime, modularity and ultra-fine granularity realtime. LiquiMedia satisfies these principles with a unique combination of the design aspects introduced in Chapter 2, which support one another in creating a novel multimedia operating system architecture.

First, LiquiMedia combines hierarchical partitioning with asynchronous message passing IPC to satisfy the principle of partitioning. The parent RMS operating system required by hierarchical partitioning imposes a rigid partition between realtime and non-realtime code while asynchronous IPC eliminates partition-violating priority inversions when the conductor and composer sub-operating systems communicate.

Second, LiquiMedia satisfies the principle of synchronous realtime execution with hierarchical partitioning, performer tasks and global time-triggered externalization. Hierarchical partitioning enables different task and scheduling models for performers and composers. All performers are synchronized with one another and with a global clock. Only by writing and submitting a performer can an application developer have code execute in realtime. Consequently, developers cannot write realtime code without explicitly acknowledging that it operates within an isochronous invocation framework that is bound to external time.

Third, LiquiMedia satisfies the principle of modularity through a combination of statistical admission control, distributed scheduling and hierarchical partitioning. The principle of modularity requires that LiquiMedia support the execution of independently authored realtime and non-realtime applications.

Supporting independently authored non-realtime applications is easy. Thanks to hierarchical partitioning, the LiquiMedia architecture can include a general purpose non-realtime child operating system. Most such operating systems, including UNIX variants, WindowsNT and VMS, support the execution of independently authored non-realtime applications.

Supporting independently authored realtime applications is more difficult. Satisfying the realtime constraints of independently authored realtime code requires an admission control mechanism. Admission control mechanisms need to know the running times of realtime applications. But, independently authored realtime applications cannot be trusted to accurately represent their resource needs — particularly if an accurate representation excludes the application from executing at all.

LiquiMedia solves this dilemma by using statistical inference to predict the running times of performers. However such calculations are expensive. Using distributed scheduling separates the production of a schedule from its use for the dispatching of tasks and so satisfies the principle of modularity without jeopardizing the conductor's efficiency.

Finally, LiquiMedia satisfies the principle of ultra-fine granularity realtime with the statistical admission control, performer and distributed scheduling design aspects. As shown in Section 6.4, performer-style tasks are at least 9.48 times more efficient than threads despite the overhead of the conductor's admission control facilities. Instantaneous admission control permits LiquiMedia to provide an efficient and almost impermeable partition between performers. Distributed scheduling efficiently supports large numbers of performers.

Per-performer dispatch overhead is $O(1)$ when running a fixed set of realtime tasks. Schedule creation and modification costs in LiquiMedia are high but this is not a problem for two reasons. First, in actual usage situations, infrequent schedule management operations are amortized over many basic periods of performer dispatch. Second, schedule management operations are most frequent during application startup. Here, dynamic linking and demand-paged loading dwarfs the time cost of schedule management.

LiquiMedia combines the following design aspects: hierarchical partitioning, global time-triggered externalization fixed performer-style tasks, distributed scheduling, statistical admission control and atomic task IPC. These design aspects work together to let LiquiMedia implement the fundamental principles of a multimedia operating system.

4.12 Summary

Unlike the solutions discussed above, LiquiMedia's novel combination of design aspects satisfies all four of the fundamental principles. First, LiquiMedia combines hierarchical partitioning, performer-style tasks and atomic task IPC to satisfy the principal of partitioning. Second, it satisfies the principle of synchronous realtime execution with globally externalized performer-style tasks. Third, LiquiMedia satisfies the principle of modularity with its admission control mechanism. Rather than trusting independently-authored performers to correctly declare their running times, LiquiMedia estimates a performer's running time from its past execution and uses this estimate to accept or reject the scheduling of a performer. Lastly, LiquiMedia satisfies the principle of ultra-fine granularity realtime with its use of the light-weight performer and distributed scheduling design aspects. Only distributed scheduling performer-style tasks permit an operating system to efficiently interleave the operation of a large number of tiny realtime fragments. Other operating systems either provide fine granularity realtime execution or support independently authored realtime code but not both. Only LiquiMedia satisfies both principles at once.

Chapter 5

Implementation

The previous chapter provided an overview of LiquiMedia's architecture. This chapter discusses the implementation of the `liqui` prototype. I built `liqui` as an experimental platform to test LiquiMedia's central features: fine grain realtime fragments, statistical admission control and feedback-driven schedule refinement.

5.1 Extending Solaris

Building `liqui` on top of Solaris 2.6 reduced implementation time and provided more time for testing the prototype's implementation of the four principles of a multimedia operating system. While the alternative of developing for the naked hardware is an interesting challenge, re-implementing code such as an RMS scheduler or thread library adds no academic contribution. Instead, LiquiMedia uses the Solaris static priority realtime facilities to provide its rate monotonically scheduled (RMS) parent OS and Solaris time-shared UNIX processes for its composer operating system.

The Solaris 2.6 kernel has the preemptive thread task abstraction [Sun95b]. The Solaris documentation calls these threads *light weight processes* (LWP).[1] Multiple LWPs may share a single set of memory mapping resources.

The kernel schedules LWP execution with a hierarchical scheduler. Solaris reserves the top half of the available priority levels for realtime LWPs. These LWPs have the *realtime scheduling class*. Solaris provides strict RMS schedule discipline for LWPs using these priority levels. The kernel does not provide priority inheritance so is vulnerable to priority inversion. A traditional UNIX decaying priority scheduler dynamically assigns the remaining priority levels to non-realtime LWPs in the *time-shared scheduling class* [Sun95c].

[1] Solaris also has a higher level wrapper for the LWP that the documentation refers to as a thread.

The kernel operates in a reentrant fashion for non-realtime LWPs. Multiple non-realtime LWPs can simultaneously block in kernel context. The situation for realtime LWPs is more problematic. Portions of the kernel must operate below Solaris's lowest realtime priorities. When combined with the kernel's lack of priority inheritance, this significantly limits the usability of Solaris's realtime threads.

Despite these limitations, LiquiMedia maps closely to the structure of Solaris. Solaris's two scheduling classes provides a hierarchical partition. The conductor executes inside a Solaris LWP with the realtime scheduling class. Further, Solaris's time-shared scheduling class and threaded task abstraction handily served as `liqui`'s non-realtime composer child OS where LiquiMedia composer threads run as non-realtime Solaris LWPs.

Building `liqui` on top of Solaris greatly reduced development effort. However, the resulting loss of low-level control imposed some practical difficulties. The most significant issues were interrupt handling, timing variance, Solaris's absence of priority inheritance and sharing access to hardware devices between realtime and non-realtime LWPs.

Many hardware devices have maximum service latencies that are impossible to handle using performer-polling. Consequently, as discussed in 4.3, LiquiMedia permits interrupts to preempt the conductor. However, interrupt handling code poses a difficult problem: it steals execution time from performers. There are two possible responses. First, include the running time of interrupt handling code in a a performer's execution time. Second, pause the counters used to measure a performer's running time during the execution of interrupt handling code.

Solaris does not permit implementing the second mechanism thought it might be possible to alter the Solaris interrupt dispatch code to implement this mechanism. Consequently, LiquiMedia includes the contribution of interrupt handlers in performer running times. Chapter 6 shows that even a performer with a constant running time has a sizable execution time variance. Interrupts cause some of this variance with rest caused by cache misses and TLB misses. The per-performer execution time variance limits `liqui`'s maximum firmness. A version of LiquiMedia running on the naked hardware would support greater firmness levels.

Not only is Solaris incapable of pausing the performance registers during interrupts but it does not even provide a mechanism by which the conductor can efficiently disable interrupts. Consequently, because it is always enabled, the basic period interrupt can arrive at any point in the conductor's execution. As a result, the conductor in `liqui` must handle all of the reentrant cases identified in B.3.7.

Solaris 2.6's lack of a priority inheritance mechanism also posed an implementation challenge. The `liqui` conduit library requires mutex-protected access to conduit data structures. These mutexes are shared between non-realtime application LWPs and the realtime conductor LWP. Consequently, a naive implementation of `liqui` experiences a priority inversion if the basic period interrupt occurs while a composer thread holds the conduit mutex. The conduit library solves this problem with a priority inheritance mechanism.

Interfacing with devices was the final implementation challenge imposed by Solaris. Portions of the Solaris kernel execute at a lower priority than LWPs in the Solaris realtime priority class. Consequently, use of kernel services such as those necessary to handle a breakpoint can also cause a priority inversion. Unfortunately, the system documentation inadequately describes the priority inversion safety of the various kernel services and device drivers.

Fortunately, it is possible to access both the sound device and the low level framebuffer device in realtime. Realtime access to these devices is sufficient for performers to generate multimedia streams for both the modalities of vision and hearing. Realtime access to the framebuffer device also enabled `liqui` to use the vertical retrace interrupt for its basic period interrupt.

The vertical retrace is an ideal source of basic period interrupts for three reasons. First, the interrupt is isochronous. Second, the rate of the interrupt (by design) exceeds the minimum rate required for a human's visual modality to fuse separate video frames into a continuous visual stream [Wic84]. Third, use of the interrupt automatically synchronizes performer execution with the refresh of the display.

Building `liqui` on Solaris sacrificed low-level hardware and operating system control. This loss of low-level control caused the implementation difficulties described above. Despite this, building `liqui` on top of an existing operating system was the correct decision for three reasons.

First, by building on top of an existing operating system, I finished coding more quickly. All modern operating systems contain large amounts of common code such as number to string conversion, stack frame setup and context switching. These functions are not academically interesting and yet require correct implementation for the operating system to function. Consequently, `liqui` builds on the facilities provided by Solaris rather than re-implementing these necessary but uninteresting functions.[2]

Second, implementing LiquiMedia as an extension to Solaris provides a better demonstration of the principle of processor partitioning than building an operating system on naked hardware. The principle separates realtime and non-realtime code by hosting them in independent sub-operating systems. Successfully combining independently-developed sub-operating systems — Solaris and `liqui` for example — more convincingly shows the principle of partitioning's strength: separation of different independent scheduling domains.

Third, LiquiMedia's admission control mechanism must operate correctly despite background system activities that increase the variance of performer execution. Solaris adds considerable variance to performer execution, compared to a native implementation of LiquiMedia. Consequently, building `liqui` on Solaris *more* rigorously tests LiquiMedia's scheduling techniques than does a native implementation.[3]

[2] In retrospect, it might have been better to implement the prototype as an extension to Linux. However, at the time that I began the implementation, the Linux kernel lacked preemption.

[3] A prototype running directly on the underlying hardware could address this with artificial sources of

Building `liqui` as an application on top of Solaris simplified the implementation. Further, augmenting Solaris actually resulted in a superior demonstration of LiquiMedia's value by showing the importance of the principle of processor partitioning and the admission control mechanism's robust handling of performer variance.

5.2 Implementation Structures

This section describes various implementation challenges encountered while implementing `liqui`, their solutions and the limitations that remain.

5.2.1 Conduits

Conduits are asymmetric non-blocking FIFOs as described in Section 4.6. `liqui`'s conduits can contain only objects that can be written in a single assembly language statement. This restriction bounds the time needed to read or write a conduit. `liqui` requires this bound because all conduit accesses occur inside a mutex required to insure that conduit position and current value are accessed atomically. Preprocessor macros and an abusive use of C's type casting facilities makes conduits generic. Section 7.2.3 discusses some possible improvements for the conduit mechanism.

5.2.2 Exception Handlers

`liqui` has a structured exception handling mechanism. The exception handling mechanism unifies the delivery of UNIX signals and LiquiMedia-specific errors to performers and composers. I implemented this mechanism for three reasons. First, the priority inheritance mechanism required centralizing UNIX signal handling. Second, the exception framework permits asynchronous termination of a Solaris LWP. Finally, the exception framework simplified error handling in `liqui` applications.

`liqui` includes a a priority inheritance system to eliminate priority inversion deadlocks between composer threads and the conductor. The priority inheritance system captures all asynchronous UNIX signals to insure that an interrupted composer thread clears the conduit mutex before the conductor attempts to acquire it. Consequently, `liqui` routes all asynchronous signals including the basic period interrupt through the priority inheritance system. `liqui` accumulates signal notifications and uses the exception mechanism to dispatch them synchronously.

variance. However, a production OS running real task loads provides a more realistic source of execution variance.

Solaris provides no easy way to asynchronously halt the execution of a Solaris LWP. Consequently, all composer threads in `liqui` run under an exception dispatch wrapper. This permits `liqui` to terminate all the threads comprising a single application without restarting. The exception framework also safely terminates an application thread if it generates a synchronous fault.

Finally, the structured exception mechanism simplified the development of applications. With an exception handling framework in place, much of the error checking needed in traditional UNIX C coding became unnecessary. Moreover, the structured mechanism provided an error detection and handling mechanism consistent across both realtime and non-realtime code.

5.2.3 Applications

`liqui` cannot execute normal Solaris applications. Instead, each application consists of a shared library containing its performers and composers. `liqui` initiates an application by having the dynamic linker load the application's shared library and then invoking the `app_main` entry point in a new composer thread.

The Solaris dynamic linker provides on-demand loading of an application's components. `liqui` fully resolves an application's dependencies on load so that a performer's first invocation does not invoke the dynamic linker.

Both the principles of partitioning and modularity require memory protection boundaries between applications. LiquiMedia's scheduling approach is compatible with traditional hardware-enforced memory protection boundaries. However, the overhead of hardware memory protection is incompatible with the principle of ultra-fine granularity realtime fragments.

Consequently, I designed `liqui` with the goal of extending it into a realtime Java Virtual Machine (JVM) where the JVM's software memory protection mechanism would provide memory protection boundaries between applications.[4] Anticipating this extension, all `liqui` application components operate inside a single shared memory space. However, while a violation of the principle of modularity, I wrote most of `liqui`'s test applications in C because this simplified their development.

5.2.4 Memory Management

LiquiMedia must reclaim memory allocated by applications with an automatic storage management for two reasons. First, LiquiMedia cannot rely on independently-authored code to

[4] As discussed in Appendix D, LiquiMedia Inc. used `liqui` to build a prototype of such a JVM that provides realtime execution of independently-authored Java code fragments.

include the function calls needed to relinquish previously allocated storage so satisfying the principle of modularity requires automatic storage management. Second, the conductor can suspend a performer midway through its execution and a suspended performer's heap allocations can only be recovered by an automatic storage management system.

Two techniques exist to automatically recover previously allocated storage in a shared heap: reference counting and garbage collection. Reference counting is incompatible with the principle of partitioning because reference counting requires every access to heap memory to contend for a lock on the count. Consequently, LiquiMedia must use some form of garbage collection.

Garbage collection does not solve all storage reclamation problems in a multi-threaded environment. A conservative collector thread must implement a write barrier while tracing in-use storage [JL96, Wil96]. An incremental collector must implement a read barrier on the heap. Fortunately, the performer task abstraction simplifies the implementation of either kind of barrier and provides a well-defined structure for root finding.

On a single processor machine, running the collector from the conductor provides the appropriate read or write barrier. Moreover, the atomic execution property of the performer task abstraction insures that the root set consists only of the stack contents of the (suspended) composer threads and any per-application writable static space.

Writing an incremental garbage collector is beyond the scope of this thesis. Consequently `liqui` allocates memory with an instrumented version of the allocation interfaces defined by the Boehm-Demers collector [Boe04]. The Boehm-Demers collector in fast-halt mode can run immediately after conductor execution in each basic period. However, `liqui` does not completely implement this feature because its peak heap size did not justify the additional development effort.

5.2.5 Timing and Measurement

LiquiMedia requires an efficient and accurate timer. The conductor reads the current time twice for each performer invocation. Consequently, satisfying the principle of ultra-fine granularity requires that a timer access have as small an overhead as possible. The conductor computes each performer's statistical profile from the values of these timer accesses. Consequently, an inaccurate timer invalidates LiquiMedia's admission control mechanism.

To simplify the implementation, `liqui` uses Solaris's `gethrtime` function to measure performer execution time. The `gethrtime` function returns the number of nanoseconds since system boot. As shown in Section 6.4, the 545ns running time of `gethrtime` is sufficiently small to satisfy the principle of ultra-fine granularity realtime.

5.3 Implementation Strategy

`liqui` is divided up into components: structured exception handling, performer objects, conduits, schedule graph management, the scheduler, the conductor, statistics functions, message logging and application loading. Each component was tested in isolation before integration into the `liqui` system.

Each core component has a matching test harness and unit tests. The harnesses were necessary because many of the components are tightly coupled. For example, as discussed in Appendix B, the scheduler has a complex interaction with the conductor. Consequently, each test harness implemented enough of a component's dependencies to permit unit testing the component in isolation.

LiquiMedia's operation is non-deterministic and so it is difficult to distinguish between outliers caused by unlikely but correct behaviour and outliers caused by programming errors. Consequently, I also performed a sequence of integration tests to show that the components operating in combination implemented the necessary architectural features from Appendix B. Appendix F lists these tests.

Each integration test had the same structure: a testing composer established an initial state of the system by directly modifying `liqui`'s internal data structures and then mutated the system. Test results were verified manually. Most integration tests were executed under a debugger with the conductor configured to run a fixed number of basic periods in non-realtime. In retrospect, automatic regression testing would have been helpful.

5.4 The Scheduler Simulator

I implemented a simulator for LiquiMedia's non-realtime scheduler. The simulator executed the scheduler's average-context admission control algorithm and numerical algorithms on a wide-variety of test schedules. All schedules were comprised of standard test performers. Each schedule's statistical profile data was sampled from `liqui`'s per-performer execution traces. Consequently, even the synthetic schedules had statistically possible and realistic execution traces.

The simulator evaluated portions of LiquiMedia that do not execute in realtime. Consequently, the simulation neither has nor requires time-accurate operation. It simply takes statistical profile data and performs admission control decisions in an identical fashion to the actual scheduler.

The simulator provides a superior evaluation of the scheduler than is possible with `liqui` for two reasons. First, because it does not operate at the behest of a realtime conductor, it can admission-test many more schedules per unit time than `liqui`'s implementation of the

scheduler. Second, the simulator can test scheduler operation under conditions that cannot be reproduced in realtime.

5.5 Summary

This chapter has described the implementation of the LiquiMedia prototype `liqui` and simulator. `liqui` is implemented above Solaris for two reasons. First, building on Solaris economized the implementation effort by requiring only the development of LiquiMedia's unique contribution. Second, building on Solaris showed that existing operating systems can be retrofitted with LiquiMedia-style support for multimedia applications. LiquiMedia's non-realtime scheduler is both implemented in `liqui` and separately in a simulator. The simulator permitted evaluating the scheduler's operation under conditions not reproducible in realtime.

I implemented these two components to demonstrate that the LiquiMedia architecture satisfies the four principles of a multimedia operating system described in Chapter 1. Chapter 6 presents the results of experiments performed with `liqui` and the simulator which demonstrate that LiquiMedia does satisfy the four principles.

Chapter 6

Performance Measurements

The thesis presents the architecture of the LiquiMedia operating system and its prototype implementation `liqui`. Chapter 4 reasoned that any bug-free implementation of LiquiMedia's architecture, like `liqui`, satisfies the four principles of a multimedia operating system presented in Chapter 1. The performance tests discussed in this chapter confirm this reasoning: `liqui` operates successfully as designed on a range of typical cases. These tests also quantify the penalties imposed by a real implementation on a possibly unrealizable ideal.

6.1 Apparatus and Methodology

This section describes the experimental software and methodology. The performance tests can be categorized by their goal. First, most of the tests discussed below provide a pass/fail confirmation of the reasoning behind LiquiMedia's design by showing that `liqui` operates successfully on a range of typical cases. Second, the remainder of the tests measure `liqui`'s performance and compare it to previous approaches.

6.1.1 Metrics

Both kinds of tests use three important metrics. They are defined here.

Loading

: A *loading* is a fraction of a basic period. `liqui` measures performer running times in nanoseconds. Because the size of these numbers makes them cumbersome, this chapter specifies performer running times and derived quantities as loadings.

  A schedule loading is the fraction of a basic period consumed by executing the performers that comprise a path through the schedule and is the sum of each executed

performer's loading. Note that `liqui` cannot support schedule loadings of 1.0 because of the time consumed by the conductor's overhead.

Firmness P_c

Firmness is an input to the scheduler that specifies the probability that a performer's actual running time is less than the upper bound computed by the scheduler. Section B.2 provides a formal definition. For example, a pure RMS operating system provides a firmness of 1.0.

Expected Utilization

The ratio of a performer's mean execution time to the execution time allocated to it by a scheduler. Expected utilization (EU) precisely specifies the processor resource sacrificed by the provision of realtime scheduling. The LiquiMedia scheduler computes a statistical estimate stat(p) of a performer's running time so the expected utilization $u(\text{stat}(p))$ after executing performer p for i basic periods is

$$u(\text{stat}(p)) = \frac{m_{p,i}}{\text{stat}(p)}.$$

This chapter compares LiquiMedia to operating systems that allocate processor resources using the worst case execution time (WCET). In this case, the expected utilization $u(\text{wcet}(p))$ is

$$u(\text{wcet}(p)) = \frac{m_{p,i}}{\text{wcet}(p)},$$

where wcet(p) is the largest execution time of performer p in *any* invocation.

It is impossible to determine wcet(p) without knowing the probability distribution function (PDF) of performer p. Consequently, this chapter approximates wcet(p) with the sample maximum

$$\text{wcet}(p) \approx \max_{i=1,\ldots,n} t_{p,i},$$

for n invocations of performer p. This approximation has a small but ignorable probability of under-estimating wcet(p) and thereby placing the Chebyshev estimator at a disadvantage compared to the WCET estimator. However, as shown in Appendix G, the probability of under-estimating wcet(p) is less than 5×10^{-8} and so can therefore be ignored.

Note that in practice, declarative admission control requires estimating a performer's wcet(p) from an inspection of its code and such estimates normally exceed wcet(p) by a large margin (c.f. 2.6.5).

Expected utilization generalizes naturally to an entire sequence of performers from a schedule graph. Assume that all performers in the schedule graph are simple (c.f. B.4.) Then the expected schedule utilization $u(\text{stat}(\mathcal{P}))$ is

$$u(\text{stat}(\mathcal{P})) = \frac{m_{\mathcal{P},i}}{\text{stat}(\mathcal{P})},$$

where stat($\mathcal{P}$) is the LiquiMedia scheduler's estimate of the entire schedule's running time and $m_{\mathcal{P},i}$ is the average running time of the schedule. The same generalization applies equally to the WCET estimator where $u(\text{wcet}(\mathcal{P}))$ is

$$u(\text{wcet}(\mathcal{P})) = \frac{m_{\mathcal{P},i}}{\text{wcet}(\mathcal{P})}.$$

For a single performer, expected utilization is equivalent to the common use of utilization: the fraction of the processor that a task actually uses when scheduled with realtime guarantees. Whole schedule utilization is the product of two fractions: the whole schedule expected utilization and the utilization penalty imposed by how the operating system schedules tasks. For example, RMS scheduling imposes a .693 penalty beyond the whole schedule expected utilization of the WCET estimator.

6.1.2 Overview of Tests

The tests can also be categorized by the experimental procedure and the software used: realtime tests use the `liqui` prototype, non-realtime tests use the LiquiMedia simulator and threaded code runs on Solaris. Realtime tests evaluated the operation of the conductor child operating system. Non-realtime tests evaluated the operation of the scheduler. Threaded experiments provided comparison data.

All realtime tests shared a similar structure. A non-realtime composer thread running in the `liqui` prototype configured and submitted one or more performers to the scheduler. The prototype collected per-performer and per-conductor running times in each basic period of execution. Both the conductor and scheduler logged their operation to a debugging trace file.

Non-realtime experiments used the LiquiMedia simulator to evaluate the operation of the scheduler. The simulator generated a stream of statistical profile information identical to the information collected by the conductor. For each basic period, the simulator chose a performer's running time at random from samples collected in realtime tests. It also computed whole schedule running times from per-performer results. Each experiment then consisted of the simulator running the relevant portions of the scheduler on the previously generated statistical profile information.

Threaded tests consisted of simple test programs running on top of Solaris without any `liqui` code. These programs measured the cost of context switching between realtime threads on Solaris and the overhead of the execution counter hardware.

6.1.3 Test Performers

The most important piece of experimental apparatus is the set of test performers because both the realtime tests and the non-realtime tests use execution information produced by them.

There are two kinds of test performers: synthetic test performers and actual multimedia audio-processing performers. From the scheduler's vantage, a performer is merely a task that consumes processor cycles and the purpose for which the performer consumes these cycles is irrelevant. Consequently, synthetic test performers serve equally well in testing LiquiMedia as do performers which generate multimedia streams and have the additional advantage of providing more repeatable results with less development effort. As a result, I generated most of the experimental data discussed herein by executing a standard set of synthetic test performers.

The scheduler is independent of the statistical distribution of a performer's running time because, as defined in Section B.2.1, it uses Chebyshev's inequality to compute the running time allocated to a performer:

$$\text{stat}(p) = \mu_p + \sigma_p \sqrt{\frac{1}{1 - P_c}}.$$

The equation shows that stat(p) is a function of only the mean of p, its standard deviation and the desired firmness P_c. In particular, stat(p) is independent of the probability distribution function of performer p.

Moreover, the expected utilization $u(\text{stat}(p))$ achieved by this allocator is a function of the ratio of the standard deviation to the mean and the firmness. Substituting the sample statistic definition of stat(p) from Equation 6.1.3 into the definition of $u(\text{stat}(p))$ makes this clear:

$$u(\text{stat}(p)) = \frac{m_{p,i}}{m_{p,i} + s_{p,i}\sqrt{\frac{1}{1-P_c}}} \tag{6.1}$$

$$= \frac{1}{1 + \frac{s_{p,i}}{m_{p,i}}\sqrt{\frac{1}{1-P_c}}} \tag{6.2}$$

Because the performer expected utilization $u(\text{stat}(p))$ is completely determined by a performer's firmness and the ratio $s_{p,i}/m_{p,i}$, LiquiMedia provides a flexible trade-off between firmness and expected utilization. Figure 6.1 shows a plot of this trade-off at three different ratios of the standard deviation to mean.

For the same reason, measuring `liqui`'s expected utilization only requires executing two synthetic test performers whose standard deviation to mean ratio spans the range of typical ratio values. I implemented the *Simple* and *Jittered Simple* test performers for this purpose.

Jittered Simple

Jittered simple loads the processor with a simple loop of floating point computations. Jittered simple executes its loop for $a + bB$ iterations where a is fixed, B is a uniform random variable on $[-1, 1]$ and b is a scale factor chosen to give Jittered Simple a larger standard deviation than a typical multimedia performer. A composer sets a and b as required.

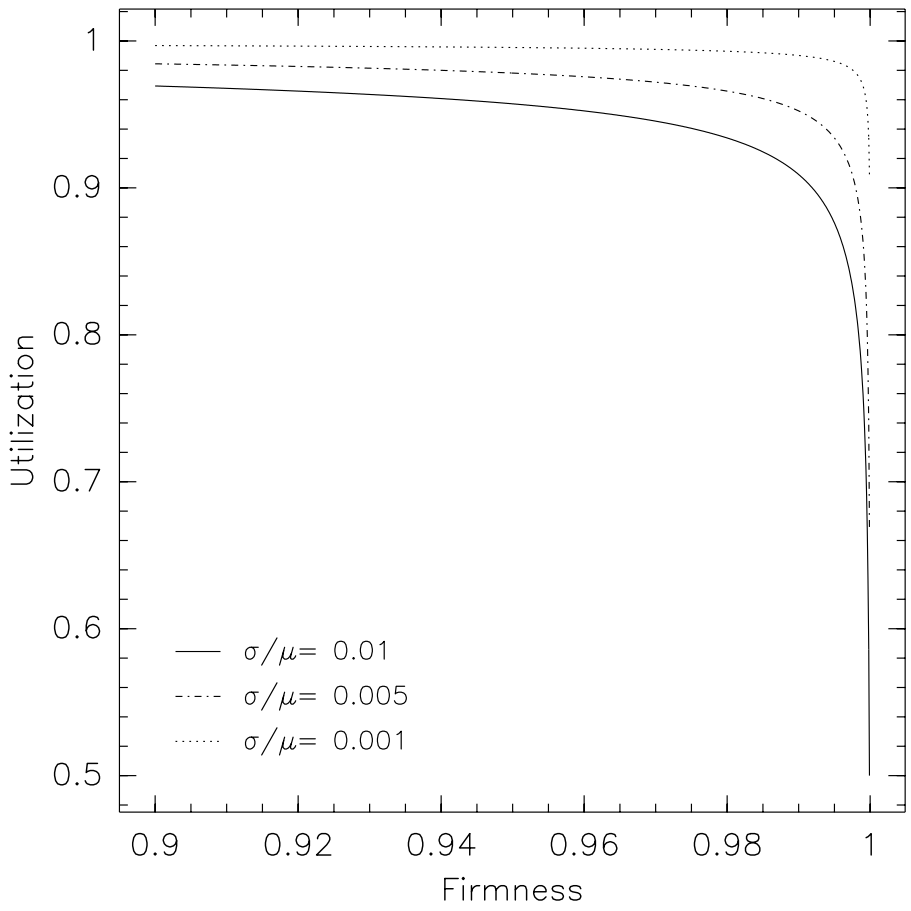

Figure 6.1: Expected utilization delivered by LiquiMedia's Chebyshev estimator as a function of firmness for performers with three different ratios of standard deviation σ to mean μ.

Simple Simple is a special case of Jittered simple where $b = 0$. Simple's running times have a theoretical standard deviation of 0 and therefore should provide the maximum possible expected utilization.

To confirm that the combination of Simple and Jittered Simple do span an adequate range of ratios, I also implemented the *Synth* performer. Synth implements a realistic multimedia load: synthesizing audio by summing a weighted combination of sinusoids into a memory buffer. A non-realtime task could set both the number of sinusoids summed and the target buffer size.

The shape of the performer's probability distribution function has no effect on the expected utilization $u(\text{stat}(p))$ achieved with the Chebyshev estimator. However, the shape of the performer's PDF does affect comparisons between $u(\text{stat}(p))$ and the expected utilization of a WCET-based estimator $u(\text{wcet}(p))$. Taking the ratio of $u(\text{stat}(p))$ and $u(\text{wcet}(p))$

$$\frac{u(\text{stat}(p))}{u(\text{wcet}(p))} = \frac{m_{p,i}}{\text{stat}(p)} / \frac{m_{p,i}}{\text{wcet}(p)} \tag{6.3}$$

$$= \frac{\text{wcet}(p)}{\text{stat}(p)} \tag{6.4}$$

$$= \frac{\max_{i=1,\ldots,n} t_{p,i}}{m_{p,i} + s_{p,i}\sqrt{\frac{1}{1-P_c}}} \tag{6.5}$$

shows why. Ignoring the contribution of the standard deviation, comparing $u(\text{stat}(p))$ and $u(\text{wcet}(p))$ actually compares the sample mean to the sample maximum. When the sample mean is close to the sample maximum, $u(\text{stat}(p))/u(\text{wcet}(p))$ is small. Conversely, when the sample mean is significantly less than the sample maximum, $u(\text{stat}(p))/u(\text{wcet}(p))$ is large.

In a continuous PDF, the difference between the sample mean and maximum depend on the skew of the performer's PDF. Performers with positive skew PDFs maximize statistical scheduling's expected utilization advantage over WCET scheduling. Figure 6.2 shows an example of two PDFs with contrasting skewness and hence differences between their means and maxima. Figure 6.2(1) is a plot of the Pareto distribution — one of a family of power distributions with large positive skew [Ree04]. Small loading instances of simple, Jittered simple and Synth, the audio performers described below and performers for MPEG decompression [AB98b] have distributions similar to the Pareto. Here, the maximum of 10 is much larger than the mean of 2.1 so the Chebyshev estimator provides much better expected utilization. Figure 6.2(2) has negative skew. Because its maximum of 10 is only slightly larger than its mean of 9.2, the WCET estimator provides better expected utilization.

To confirm the impact of distribution shape on EU, I implemented the pathological test performer *Sinusoidal*. Sinusoidal is non-Pareto, has negative skew, is unlikely in practice and is nearly an inverse of distributions optimal for LiquiMedia. Sinusoidal completes the set of test performers by providing a test performer with an atypically difficult PDF.

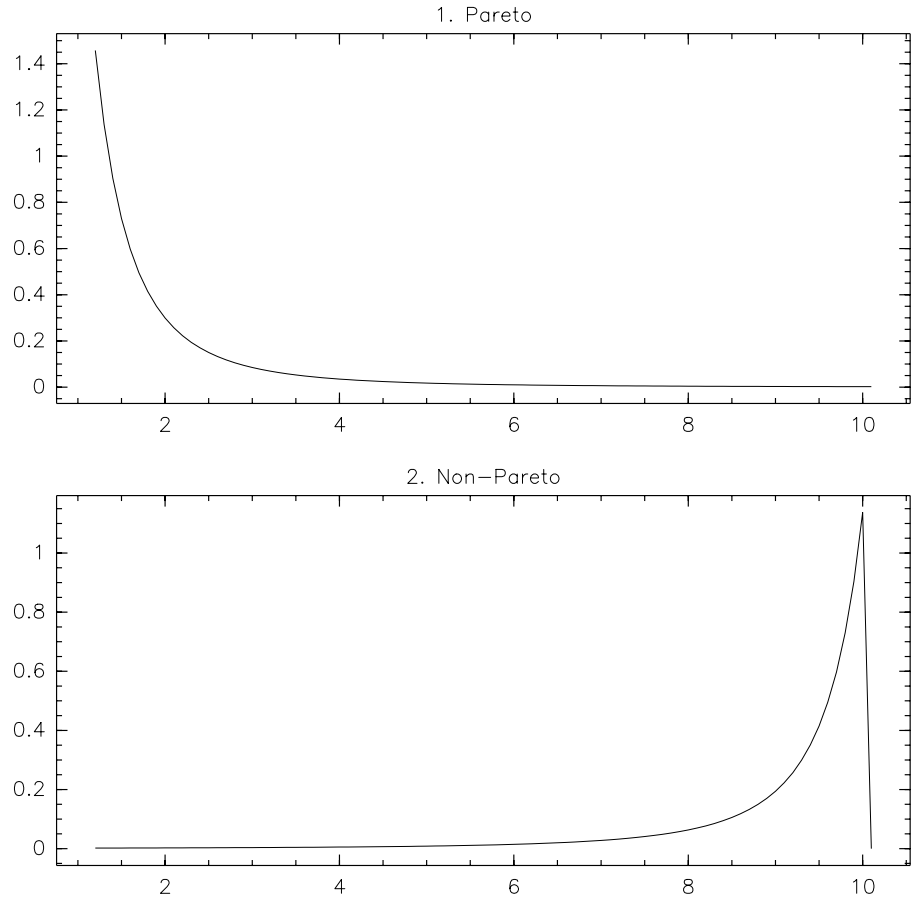

Figure 6.2: The upper graph shows a Pareto distribution. This distribution's large positive skew causes statistical scheduling to have a significant EU advantage over WCET scheduling. The lower graph shows an "opposite" distribution where negative skew gives WCET scheduling better expected utilization.

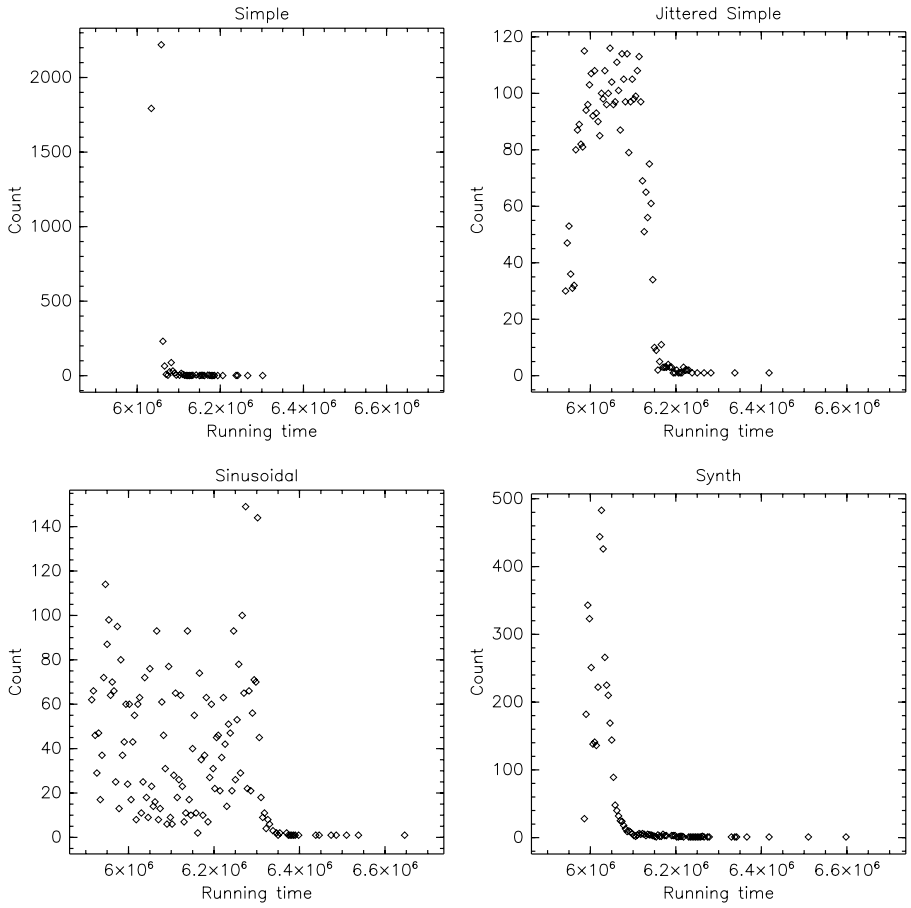

Figure 6.3: Running time distribution of the standard test performers at a 0.4 loading level.

Sinusoidal has a load loop identical to Jittered simple but executes for $a+b\sin(ci)$ iterations where i is the basic period, a is fixed, b is a scale factor and c adjusts the period. As with Jittered simple, a composer task sets a, b and c. Besides having negative skew, Sinusoidal also provides a worst case EU bound for `liqui` by having more than twice the standard deviation of Jittered.

Summarizing, the performance tests used four different synthetic performers. As shown in Figure 6.4, Simple, Synth and Jittered span the range of typical standard deviation to mean ratios The histograms of the four synthetic test performers at the 0.4 loading level found in Figure 6.3 show how the distribution shapes of the synthetic test performers differ. Simple, Synth and Jittered have distributions similar to Figure 6.3(1). Conversely, Sinusoidal provides a lower bound on `liqui`'s expected utilization and advantage over WCET estimation because, as shown in Figures 6.4 and 6.3, it has both a much higher standard deviation to mean ratio

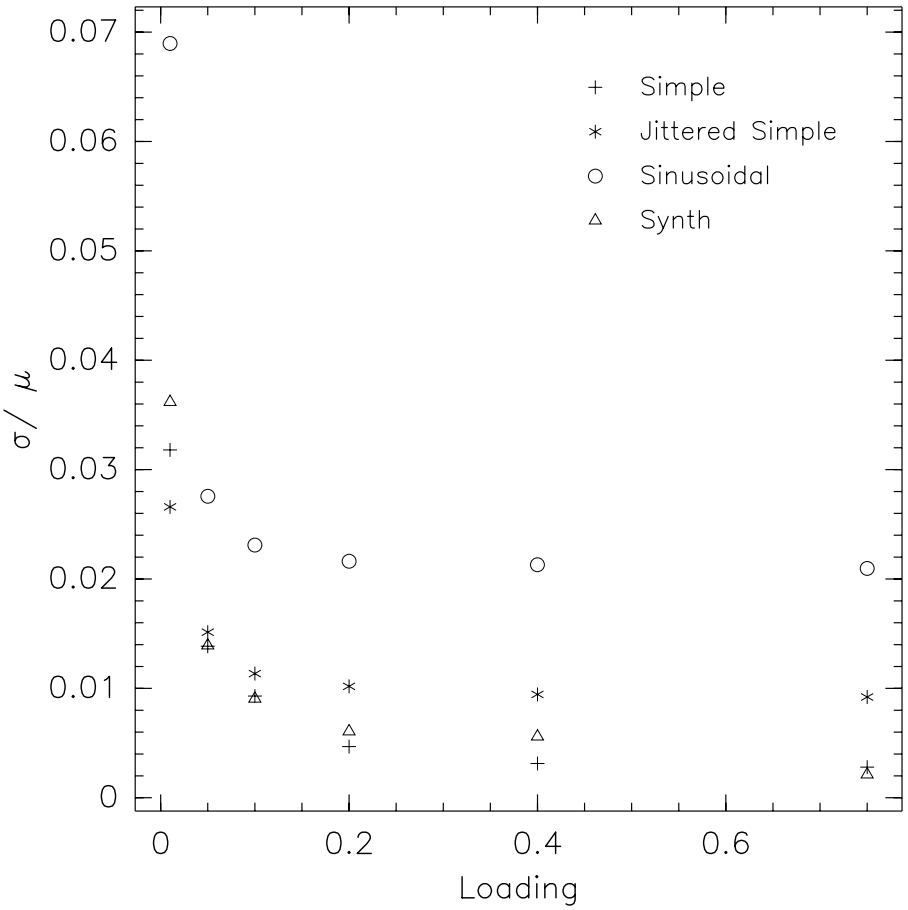

Figure 6.4: Standard deviation to mean ratios of the synthetic test performers.

and a distribution much more like the pathological case of Figure 6.3(2).

I also tested LiquiMedia with the `player_8` and `FAP` multimedia performers from the LiquiMedia audio player described below in Section 6.1.4. `player_8` performs a multi-pole filter on a sequence of blocks of audio samples. `FAP` delivers filtered blocks to the Solaris audio driver. Figure 6.5 shows the running time distributions of these two performers.

6.1.4 Realtime Test Details

Three kinds of realtime tests used these test performers: basic execution experiments, the peak loading test and the audio player experiments. These tests showed that the conductor operated as designed, measured its performance and provided data for the simulator. The tests are described below.

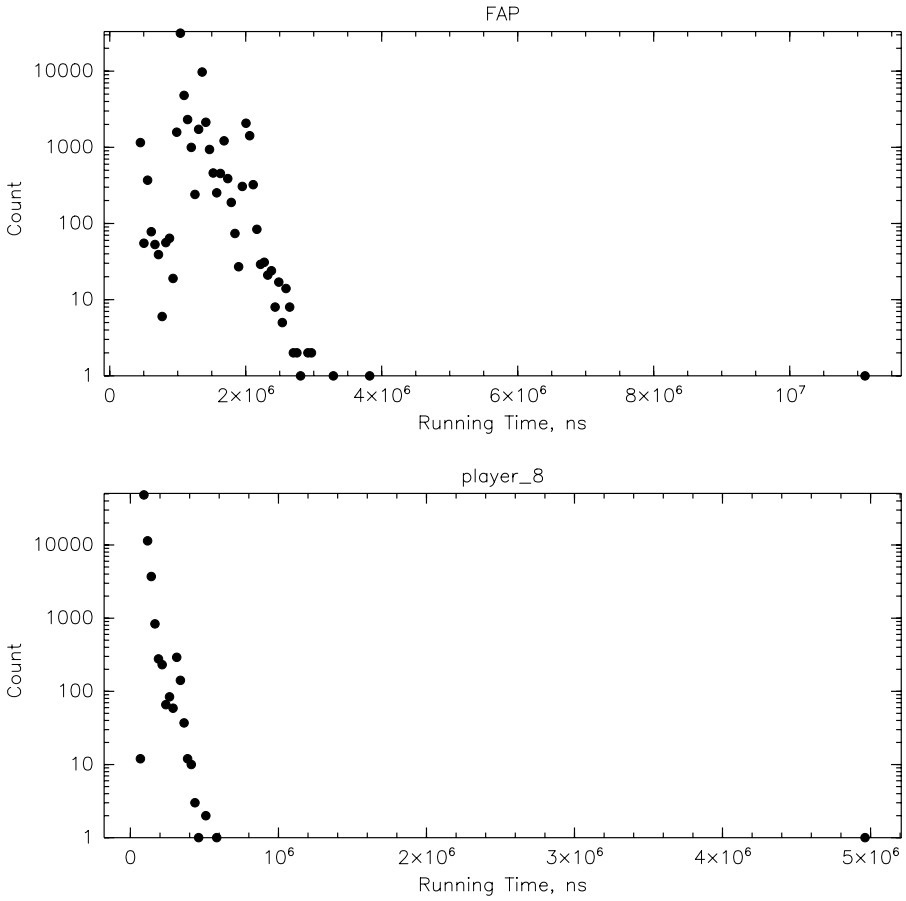

Figure 6.5: Log histogram of the running times of the audio player performers.

Basic Execution Tests

Basic execution tests consisted of running single instances of one of the four synthetic test performers described in Section 6.1.3 at six different estimated loadings for one minute: 4561 basic periods of the `liqui` prototype's execution. The six different loadings were 0.01, 0.05, 0.10, 0.20, 0.40 and 0.75 of the basic period.

Peak Loading Test

The peak-loading experiment provided the results needed to compute a performer's running time overhead. Two versions exist: the performer-based peak-loading experiment and the threaded peak-loading experiment.

In the performer-based experiment, a composer added blocks of 5 simple test performers with a 0.01 loading to the schedule every 300 basic periods (approximately 4 seconds) until the lifetime admission control mechanism rejected the candidate schedule at an attempted load of 95 performers.

That the scheduler rejected the candidate schedule at 95 shows the presence of its lifetime admission control mechanism. Figure 6.9 shows that the per-performer expected utilization is approximately .9 when the schedule contains 90 instances of 0.01 simple. Consequently, the lifetime admission control mechanism rejected a candidate schedule containing 95 performers because this candidate has a Chebyshev-estimated running time of 1.06 of the basic period. The lifetime admission control mechanism reduced the schedule to 89 performers during a schedule re-verification when the size of schedule's standard deviation caused the 90 performer schedule to be inadmissible.

The threaded peak-loading experiment provided the same data for a thread's running time overhead. In the threaded peak-loading experiment, creative use of mutual exclusion primitives forced a fine-grain interleaving of a fixed number of fixed-load threads. Each thread repeatedly acquired the mutex, executed a synthetic load derived from the simple test performer and released the mutex. The software recorded the total execution time and the per-iteration thread running time.

Audio Player

The audio player experiment consisted of monitoring a stream of audio produced by an audio player application while the conductor recorded per-performer and total conductor running times. The audio player was a Java application running inside a realtime Java virtual machine (JVM) built by LiquiMedia Inc. to run on top of the `liqui` prototype. Appendix D describes LiquiMedia Inc. and the realtime JVM in further detail.

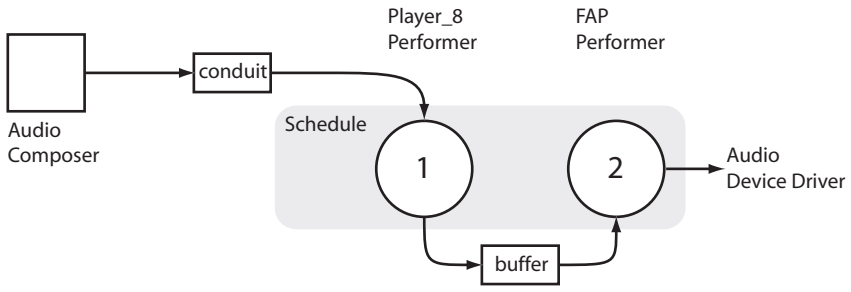

Figure 6.6: Architecture of the audio player application.

Figure 6.6 shows the structure of the audio player. A non-realtime audio composer generated stereo 8kHz PCM audio samples and wrote them individually to a conduit. The Java `player_8` performer filled a buffer by sequentially reading individual samples from the conduit. It adjusted the buffer size to keep the audio playback synchronized with basic period boundaries. Then, it filtered the block of audio samples and passed the block to the `FAP` performer. The `FAP` performer wrote the audio sample blocks to the underlying audio hardware.

6.1.5 Summary

Performance testing consisted of three kinds of tests: realtime tests to evaluate the conductor, non-realtime tests to evaluate the scheduler and threaded tests to provide comparison data. Both the realtime and non-realtime tests relied on four standard test performers: Simple, Jittered, Synth and Sinusoidal. These test performers exhibited running time distributions and sample statistics ranging from the ideal Simple through the typical Jittered and Synth to the pathological Sinusoidal. Tests also used the audio-processing performers `player_8` and `FAP`. The following sections each discuss experimental results and how they demonstrate that `liqui` satisfies the principles of a multimedia operating system.

6.2 Partitioning

A hypothetical perfect implementation of the LiquiMedia architecture implements the principle of partitioning by design. The `liqui` prototype has imperfections including its dependence on Solaris and undiscovered bugs. The audio player tests show that, despite these imperfections, `liqui` satisfied the principle of partitioning for a moderately challenging combination of non-realtime and realtime loadings.

The audio player provides a strenuous yet easily monitored test of `liqui`'s partitioning. It is a strenuous test because it provides over 16000 opportunities per second for partitioning

failure. The conduit library is the only possible cause of partitioning failures because it is the sole gateway through `liqui`'s partition between realtime and non-realtime tasks. The audio player exercises the conduit library by passing stereo 8kHz audio through it sample-by-sample. This invocation pattern requires 16000 unique writes from a composer to the conduit where one of the conduit library's two failure cases can occur at every write.

The conduit library can fail in two ways: priority inversion or composer lock-out. I easily monitored the audio player test by listening to its generated audio stream because each of these failure modes permanently disrupt the audio stream. Priority inversion halts the machine and so stops the audio playback. Composer lock-out repeatedly plays the same one second segment of music. I could easily detect such a change while using the machine hosting `liqui` to edit code, handle e-mail and browse the web.

Twelve hours of error-free music playback on the `liqui` prototype showed that the conduit library maintained partitioning in seven hundred million successive invocations. This result is sufficient to confirm that `liqui` satisfies the principle of partitioning.

6.3 Synchronous Realtime

LiquiMedia's architecture is both synchronous and realtime by design. Consequently, it satisfies the principle of synchronous realtime. However, as with all other realtime operating systems, LiquiMedia sacrifices utilization for realtime execution guarantees. This section discusses the utilization levels achieved by `liqui` in the basic execution and audio player tests.

The measured expected utilization (EU) is compared to the expected utilization of RMS scheduling. Expected utilization can be compared in two ways: per-performer and over the whole schedule. Per-performer compares `liqui`'s per-performer EU $u(\text{stat}(p))$ with the WCET estimator's per-performer EU $u(\text{wcet}(p))$. The basic execution experiments show that LiquiMedia delivers per-performer expected utilization $u(\text{stat}(p))$ equal or better to the WCET estimator's $u(\text{wcet}(p))$ at a firmness of 0.99 for typical σ/μ ratios and loadings.

In comparing expected utilizations over the whole schedule, `liqui` always has better expected (and actual) utilization than RMS scheduling. An RMS operating system provides independently authored tasks with a firmness of 1.0 but only when all tasks' utilization does not exceed $.693u(\text{wcet}(\mathcal{P}))$.[1] `liqui`'s whole schedule expected utilization $u(\text{stat}(\mathcal{P}))$ always exceeds $.693u(\text{wcet}(\mathcal{P}))$ because $u(\text{stat}(p)) \geq u(\text{wcet}(p))$ implies $u(\text{stat}(\mathcal{P})) \geq u(\text{wcet}(\mathcal{P}))$ and synchronous execution of tasks in LiquiMedia eliminates the .693 utilization cost of RMS scheduling.[2]

[1] Better RMS utilizations are possible when tasks have similar periods. However, an RMS operating system supporting independently-authored tasks cannot rely on them having similar periods. Consequently, it is

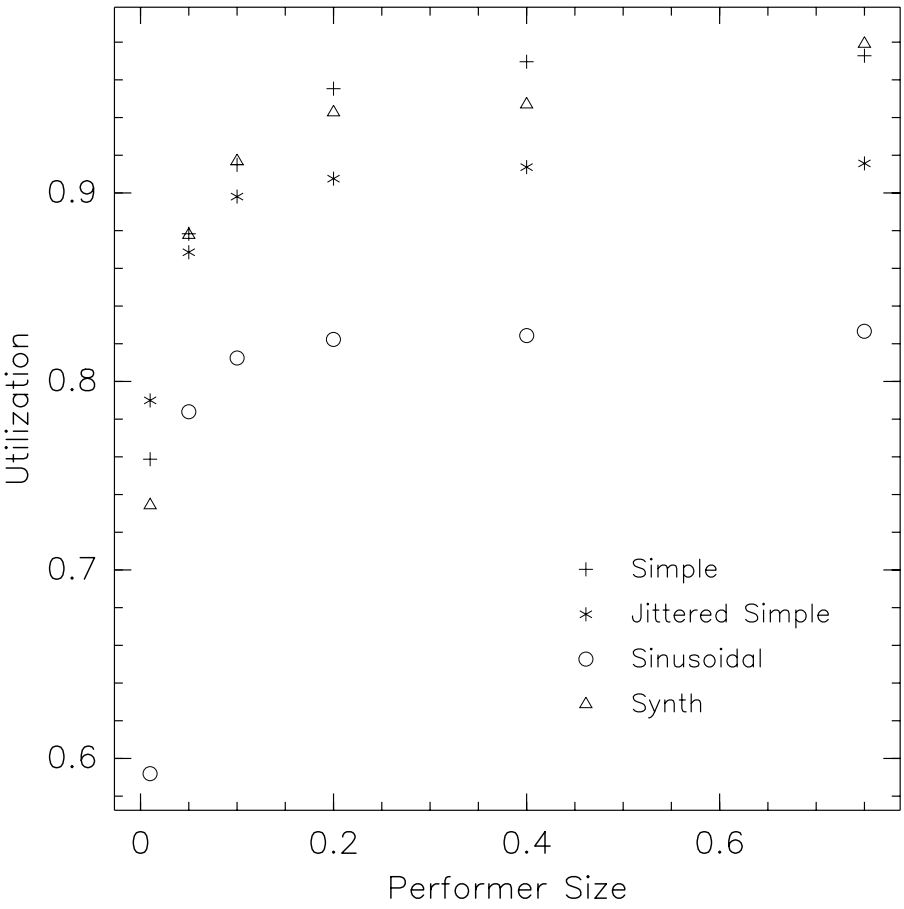

Figure 6.7: Expected utilization $u(\text{stat}(p))$ of the standard test performers at a firmness of $P_c = 0.99$. Variance caused by interrupts limit the expected utilization of smaller performers. As expected, the pathological sinusoidal performer has significantly poorer EU at all performer sizes.

Loading	Firmness	Simple	Jittered	Sinusoidal	Synth
0.01	0.95	0.876	0.894	0.764	0.861
	0.99	0.759	0.79	0.592	0.734
0.05	0.95	0.942	0.937	0.89	0.941
	0.99	0.878	0.869	0.784	0.878
0.1	0.95	0.96	0.952	0.906	0.961
	0.99	0.915	0.898	0.812	0.917
0.2	0.95	0.98	0.956	0.912	0.974
	0.99	0.955	0.908	0.822	0.943
0.4	0.95	0.986	0.959	0.913	0.976
	0.99	0.97	0.914	0.824	0.947
0.75	0.95	0.988	0.96	0.914	0.991
	0.99	0.973	0.916	0.827	0.979

Table 6.1: The table shows the expected utilization levels of the standard test performers at firmness values of 0.95 and 0.99 and six different target loadings.

Figure 6.7 summarizes the expected utilization $u(\text{stat}(p))$ of the per-performer basic execution experiments while Table 6.1 presents Figure 6.7's underlying data and adds EU values obtained at a firmness of $P_c = 0.95$. Inspecting the table shows that LiquiMedia delivers at least 0.9 expected utilization for all but the sinusoidal performer at target loadings greater than or equal to .1. The results confirm the challenge of scheduling the sinusoidal test performer.

Figure 6.8 compares the Chebyshev expected utilization $u(\text{stat}(p))$ at a firmness of 0.99 with the WCET expected utilization $u(\text{wcet}(p))$ by plotting $u(\text{stat}(p))/u(\text{wcet}(p))$ as a function of target loading for the standard test performers. Table 6.2 contains the data underlying figure 6.8. With the exception of the sinusoidal test performer, LiquiMedia's expected utilization meets or exceeds that provided by the WCET estimate.

Figures 6.7 and 6.8 also confirm the importance of the distribution shape in comparing the expected utilizations possible with Chebyshev and WCET estimators. The performers with smaller loadings have both lower expected utilizations and superior expected utilization compared to WCET estimates. The larger positive skew of the running time distributions of small loading performers explains this utilization difference.

The inclusion of interrupt handlers in performer running times causes the larger positive skew of small loading performers. Interrupts increase the variance of performers with short running times more than they affect the variance of long-running performers and consequently

necessary to compare `liqui`'s utilization with the utilization achieved by RMS scheduling of arbitrary tasks.

[2]Eliminating the RMS utilization penalty also eliminates the ability to execute tasks with different periods. Supporting tasks with different periods is unnecessary in a multimedia operating system where global synchronization is a desirable feature. Obviously, it is not an appropriate feature for all application domains.

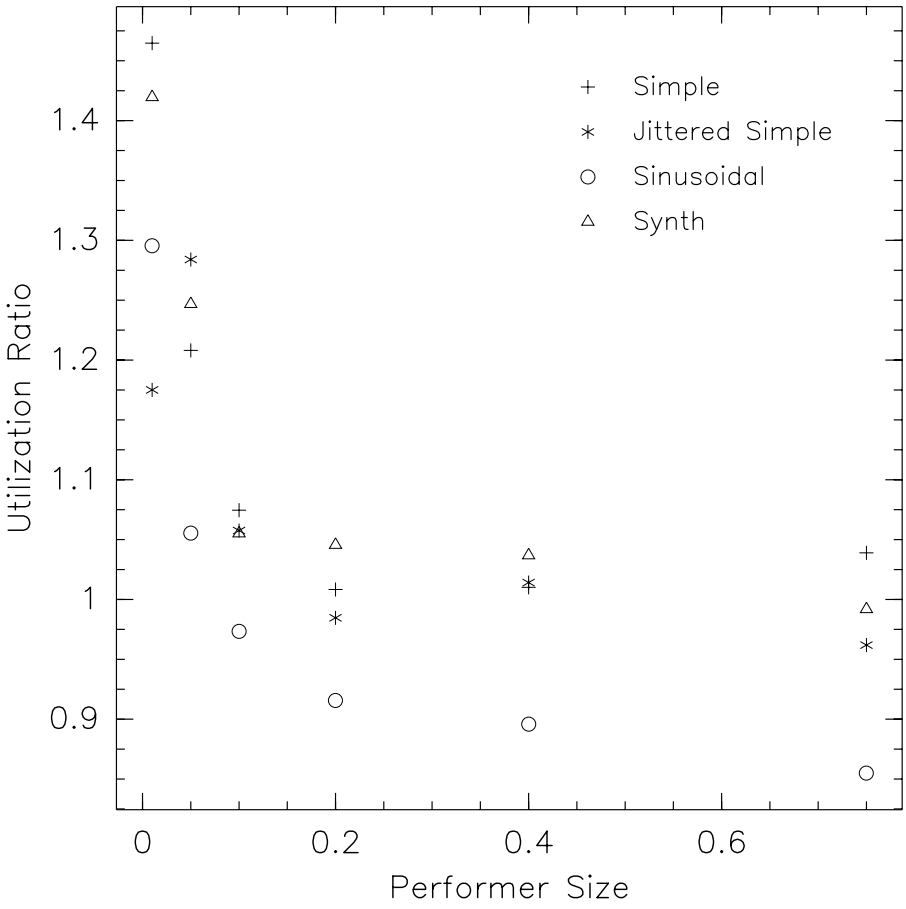

Figure 6.8: The graph compares $u(\text{stat}(p))$ with $u(\text{wcet}(p))$ for each of the standard test performers at a range of performer sizes. Only statistically scheduled performers with a sinusoidal running time distribution fail to meet or exceed the WCET bounds. Note the relative superiority of statistical scheduling for the highly variant smaller performers.

Loading	Allocator	Simple		Jittered Simple		Sinusoidal		Synth	
		util.	ratio	util.	ratio	util.	ratio	util.	ratio
0.01	WCET	0.518		0.672		0.457		0.517	
	0.95	0.876	1.7	0.894	1.3	0.764	1.7	0.861	1.7
	0.99	0.759	1.5	0.79	1.2	0.592	1.3	0.734	1.4
0.05	WCET	0.727		0.676		0.743		0.704	
	0.95	0.942	1.3	0.937	1.4	0.89	1.2	0.941	1.3
	0.99	0.878	1.2	0.869	1.3	0.784	1.1	0.878	1.2
0.1	WCET	0.851		0.849		0.835		0.869	
	0.95	0.96	1.1	0.952	1.1	0.906	1.1	0.961	1.1
	0.99	0.915	1.1	0.898	1.1	0.812	0.97	0.917	1.1
0.2	WCET	0.947		0.922		0.898		0.902	
	0.95	0.98	1	0.956	1	0.912	1	0.974	1.1
	0.99	0.955	1	0.908	0.98	0.822	0.92	0.943	1
0.4	WCET	0.96		0.901		0.92		0.913	
	0.95	0.986	1	0.959	1.1	0.913	0.99	0.976	1.1
	0.99	0.97	1	0.914	1	0.824	0.9	0.947	1
0.75	WCET	0.936		0.952		0.967		0.987	
	0.95	0.988	1.1	0.96	1	0.914	0.95	0.991	1
	0.99	0.973	1	0.916	0.96	0.827	0.85	0.979	0.99

Table 6.2: The table compares the expected utilizations achieved by the WCET and Chebyshev running time estimators based on complete sample statistics.

push the running time distribution of a small loading performer toward the positive skew Pareto distribution shown in Figure 6.2(1).[3]

The Simple performer in Figure 6.2 clearly shows the "noise floor" caused by interrupts because it executes a constant number of assembly language instructions in each invocation. Separately accounting for interrupts would remove their contribution to a performer's variance but is of dubious benefit because it does not correct for the variance added by cache and TLB misses following an interrupt handler's execution.[4] Consequently, the additional system complexity needed to separate interrupt running time from performer running time is both pointless and unwarranted because `liqui` already delivers firmness levels sufficient for multimedia stream generation.

Because embedded interrupts make the PDFs of short-duration performers resemble the distribution shown in Figure 6.2(1), the Chebyshev-style estimator delivers at least 25% greater expected utilization than the WCET estimator for synthetic test performers with loadings below 0.05. For test performers with higher-loadings, the Chebyshev-style estimator provides only a 0.05% expected utilization advantage. While low, this level of expected utilization remains acceptable because it permits `liqui` to execute independently-authored performers at a firmness of 0.99. However, if performers encountered in practice have running time distributions with negative skew such as shown in Figure 6.2(2), then support for independently-authored performers may come at the cost of impractically low utilizations.

Fortunately, realistic signal-processing performers such as a MPEG decoder have positive skew distributions similar to the Pareto PDF shown in Figure 6.2(1) [AB98b]. Also, as shown in Figure 6.5, the signal-processing performers `FAP` and `player_8` from the audio player described above in Section 6.1.4 have positive skew distributions more like figure 6.2(1) than figure 6.2(2).

Table 6.3 shows $u(\text{stat}(p))$, $u(\text{wcet}(p))$ and $u(\text{stat}(p))/u(\text{wcet}(p))$ for the audio performers `FAP` and `player_8`. For `player_8`, the Chebyshev-style estimator's expected utilization is 2.6 times better than the WCET estimator's EU while for `FAP`, the Chebyshev-style estimator provides an order of magnitude improvement in expected utilization. For such performers, the Chebyshev estimator offers significant expected utilization advantages over the WCET estimator.

[3]Recall that `liqui` is built on top of Solaris. In Solaris, interrupts come from sources such as the Ethernet hardware and keyboard. Consider an operating regime where most interrupts are caused by the 2400 baud keyboard interface. Here, interrupts occur with a probability of 1.6×10^{-5} per clock cycle and execute in about .001 of a basic period. The number of interrupts is reasonably modeled by a binomial distribution.

A performer running for 0.01 of a basic period would expect to see .4 interrupts in its execution. As a result, its most likely outcomes are either 0 or 1 interrupts. These two outcomes have a 10% difference in running time. A performer running for .75 of a basic period expects 27.7 interrupts during its execution. Such a performer would see only a a .1% difference in running times for invocations containing 27 or 28 interrupts.

[4]An interrupt that flushes a performer from a processor cache significantly increases the running time of a short-running performer. For example, each loop through simple takes 21ns on the LiquiMedia test machine when executing from cache. However, reloading the code from main memory takes 500ns. If the processing of interrupts flushes the cache for even 1% of Simple's loop cycles, its execution time will rise by 23%.

Performer	Firmness	Utilization		Ratio
		Chebyshev	WCET	
`FAP`	0.95	0.417	0.0212	19.6
	0.99	0.242		11.4
`player_8`	0.95	0.466	0.108	4.31
	0.99	0.281		2.6

Table 6.3: The expected utilizations of the audio performers using both Chebyshev-style and WCET estimators and the ratio between them.

These results are for a firmness of 0.99. Increasing the firmness reduces statistical scheduling's utilization advantage. However, a firmness of 0.99 is more than sufficient for the needs of a multimedia operating system for the reasons discussed previously in Section 4.2

As shown by the peak-loading test, `liqui` delivers better expected utilization than RMS scheduling when executing large numbers of non-pathological small-loading performers. Figure 6.9 shows the whole-schedule expected utilizations $u(\text{stat}(\mathcal{P}))$ and $u(\text{wcet}(\mathcal{P}))$ at the different load levels used by the peak-loading test. `liqui`'s whole-schedule expected utilization levels are nearly double that of a WCET estimator for two reasons. First, as shown in Table 6.2, the Chebyshev estimator provides 1.5 times better expected utilization than the WCET estimator for a single instance of the .01 simple test performer. Second, caching and improved interrupt distribution increase expected utilization of successive instances of the .01 test performer in a similar fashion to the utilization increase shown by the .05 test performer (c.f. 6.1). The overhead of overtime handling reduces utilization at the highest loading level. Poor performance at a loading of .5 is caused by the overhead of interrupts buried in performer execution.

In summary, the empirical results show that for typical test performers, LiquiMedia's Chebyshev-style estimator delivers equal or better expected utilization than the WCET estimator. Further, as shown by the small loading standard performers and the audio player performers, as performer running time distributions tend to a Pareto distribution, the Chebyshev-style estimator achieves much better expected utilization than the WCET-style estimator. Consequently, LiquiMedia can provide its inherently synchronous realtime performers with adequate utilizations at firmnesses appropriate for multi-media tasks.

6.4 Ultra-Fine Granularity Performers

The principle of ultra-fine granularity permits application developers to divide a realtime application into a large number of cooperating tasks. A highly-efficient realtime task abstraction is required to provide ultra-fine granularity realtime. Otherwise, the unavoidable overhead of switching between tasks would consume an unacceptably large fraction of the available pro-

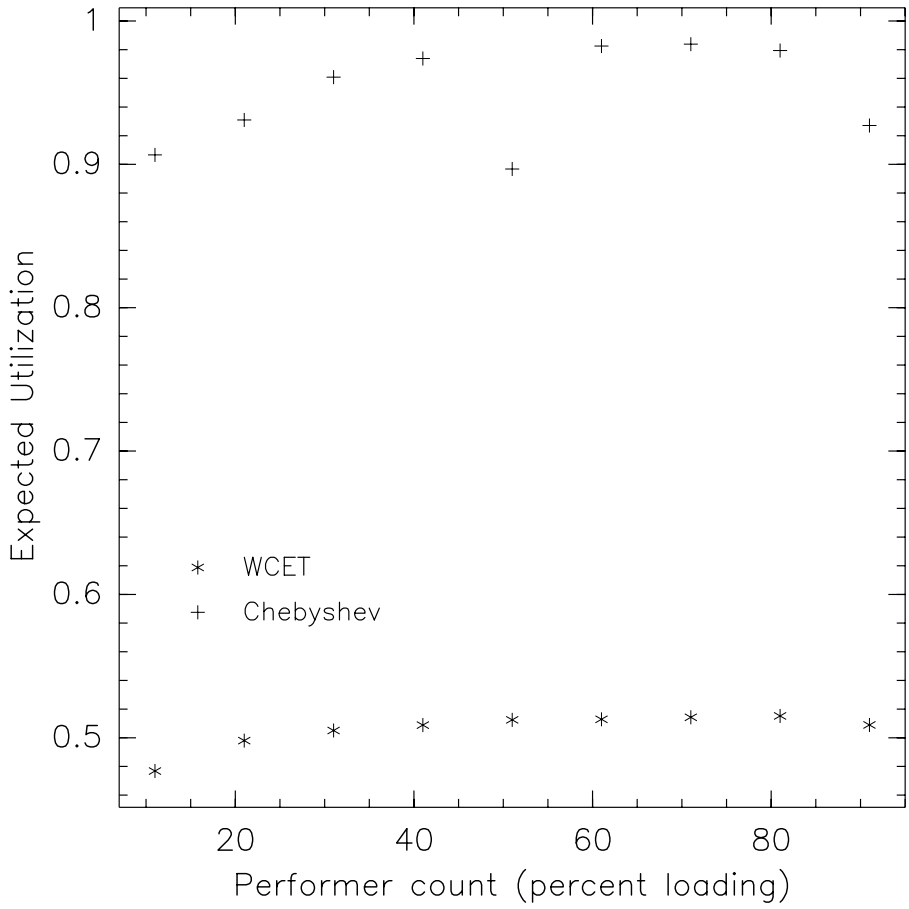

Figure 6.9: The graph compares whole-schedule Chebyshev expected utilization $u(\text{stat}(\mathcal{P}))$ with WCET expected utilization $u(\text{wcet}(\mathcal{P}))$ as a function of the number of performers in the schedule.

cessor resources. The experiments described in the remainder of this section compare `liqui`'s performer-style task abstraction with the threaded task abstraction and show that performers have the efficiency levels needed to support the repeated execution of many thousands of tasks.

6.4.1 Conductor and Performer Overhead

Conductor overhead has fixed and marginal portions:

w_p The per-performer overhead of the conductor.

w_c The conductor's fixed overhead independent of the number of performers executed.

These two constants satisfy the following equation:

$$t_i - \sum_{p \in \text{path}(1,i)} t_{p,i} = |\text{path}(1,i)|w_p + w_c \tag{6.6}$$

where $|\text{path}(1,i)|$ is the number of performers executed in basic period i, t_i is the conductor's running time in basic period i including the time it spends executing performers and $\sum_{p \in \text{path}(1,i)} t_{p,i}$ is the time spent executing performers.[5]

To reduce instrumentation overhead, the peak-loading experiment collected only the whole conductor values t_i for eight different loading levels ranging from 10 to 80 instances of the .01 simple test performer. Substitute

$$\min_{\forall i}(t_{p,i})|\text{path}(1,i)| < \sum_{p \in \text{path}(1,i)} t_{p,i}$$

for $t_{p,i}$ in Equation 6.6 to define an upper bound on w_p. Then, an upper bound on `liqui`'s per-performer overhead w_p is the slope of a linear regression fit of equation 6.7. Figure 6.10 shows the regression.

$$t_i - \min_{\forall i}(t_{p,i})|\text{path}(1,i)| > |\text{path}(1,i)|w_p + w_c \tag{6.7}$$

The linear regression provides an upper bound on `liqui`'s per-performer overhead of w_p of 1566 nanoseconds.

6.4.2 Comparison

Comparing the 1556 nanosecond bound to the overhead of threads shows the efficiency advantages of the performer task abstraction. The threaded peak-loading experiment described in Section 6.1 measured the context switch time w_f of Solaris user-space threads.

[5]Section B.4.1 defines the schedule structure and execution paths.

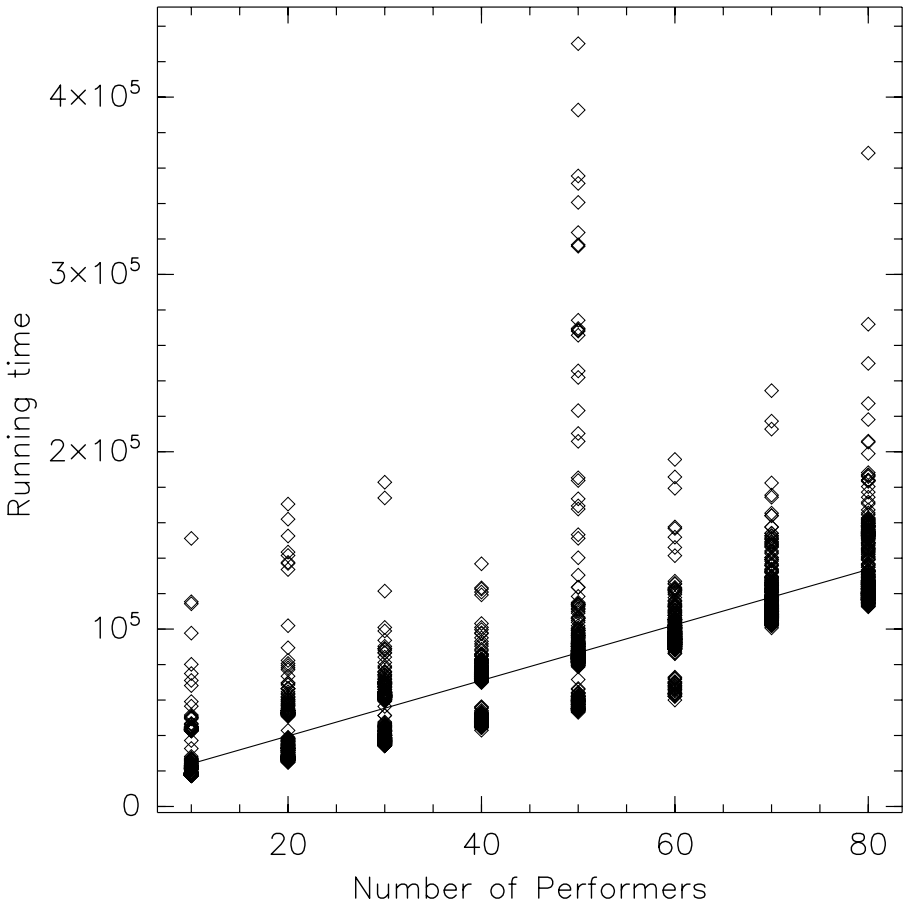

Figure 6.10: The plot shows the regression fit of equation 6.7 from data collected by the peak-loading experiment.

Metric	Threads	Performers
overhead: w_c	.00118	0.00012
Maximum number of tasks	886	8402

Table 6.4: The table compares LiquiMedia's per-performer overhead (including the profiling and admission control mechanism) with the overhead of context switching between Solaris threads. Performers are 9.48 times better.

Table 6.4 compares the normalized performer overhead w_p and the thread overhead w_f. It shows the per-task overhead as a fraction of the basic period and the maximum number of tasks possible. The maximum number of tasks in a basic period is bounded above by T_B/w_c or T_B/w_f: the number of tasks at which the operating system overhead required to dispatch them exceeds the length of the basic period.

The performer task abstraction supports 8402 tasks versus the threaded task abstraction's 886 tasks. Performers are 9.48 times more efficient than threads. However, the difference is even more significant that the results in Table 6.4 suggest. First, the performer result is only a lower bound on the maximum possible number of tasks. Second, LiquiMedia's implementation has abundant room for additional optimization whereas the thread results correspond to Sun's optimized threads library.

For example, the conductor uses the Solaris `gethrtime` function twice for each performer invocation. Converting the conductor to use the hardware performance counters lowers w_p by 1090 nanoseconds — taking the upper bound from 1566 to 476 nanoseconds. This improvement increases LiquiMedia's maximum number of tasks from 8402 to 27643 and gives the performer task abstraction a 31 times advantage over the threaded task abstraction. Additional optimizations such as re-writing the conductor in assembly language would further improve performance.

Finally, although interrupts contribute some portion of the per-performer overhead, the comparison remains fair because the running of the threaded test also includes overhead caused by interrupts.

In summary, the results show that the performer task abstraction has a significantly lower overhead than the thread-style task abstraction. This 9.48 times efficiency advantage over threads shows that `liqui` satisfies the principle of ultra-fine grain realtime.

6.5 Modularity

LiquiMedia's admission control mechanisms satisfy the principle of modularity. Three test results support this claim. First, the sample mean and standard deviations collected by the

conductor converge within the stream segregation threshold to sufficiently accurate approximations of the underlying distribution's mean and standard deviation. Second, given these accurate approximations, the instantaneous admission control mechanism eliminates all overtimes for a large number of randomly-generated schedules. Third, the lifetime admission control mechanism only admits randomly-generated schedules with a firmness exceeding the desired completion probability P_c.

6.5.1 Convergence

Successful operation of LiquiMedia's lifetime and instantaneous admission control mechanism requires that LiquiMedia's approximation to the mean and standard deviation converge within 1.5 seconds to values with better than probability P_c of approximating the true distributions' summary statistics. Further, admission testing within a stream's segregation delay requires mean values to converge within 150 milliseconds to the underlying distribution's mean.

It is impossible to determine the true running time distribution $\phi_p(t)$ of a performer p. Instead, I will use the summary statistics for a population of 4000 performer executions (53 seconds of the prototype's execution) to provide an adequate approximation to the true distribution's moments.[6] Given the number of samples, this is a reasonable assumption.[7]

Per-performer mean estimates converge rapidly to the entire sample mean. Figure 6.11 plots the normalized absolute value of the differences between LiquiMedia's running approximation to the mean $m_{p,i}$ and the actual experimental summary statistic $m_{p,4000}$. The difference is roughly bounded above by $1/2i$ so it takes at most 50 basic periods (0.7 seconds) for $m_{p,i}$ to lie within 1% of $m_{p,4000}$. In fact, a significant proportion of $m_{p,i}$ for $i > 100$ lie within .1% of $m_{p,4000}$. Moreover, means for all but .01 loading performers and Sinusoidal lie within 1% of $m_{p,4000}$ in under 10 basic periods. This convergence rate supports statistical scheduling of typical performers with a firmness of $P_c = .99$ within a user's segregation threshold.

The graphs in Figure 6.11 also show two other interesting features. First, the graphs have large y-axis spiking as the number of included samples approaches 4000. The occasional basic periods in which the performer has an unusually long running time explain these spikes. Second, as expected, convergence rates are poorest for the performers (as shown in Figure 6.4) with the largest μ/σ ratios: Sinusoidal is worst at large loadings while all performers are poor

[6] The tests ran for 4561 basic periods – one minute. Truncating that data to 4000 samples improved the appearance of the graphs.

[7] Define Z to be

$$Z = \frac{m_{p,i} - \mu_p}{\sigma_p/\sqrt{i}}.$$

Then, the central limit theorem holds that as $i \to \infty$, the distribution of Z is the standardized normal distribution $n(z; 0, 1)$ [WM78]. I used this theorem to verify the assumption that 4561 samples of a performer's execution adequately approximate its true distribution. Given $i = 4000$ and conservatively assume that $\sigma_p < 3s_{p,4000}$. Then, for the performer Jittered simple, $m_{p,4000} - \mu_p < .003m_{p,4000}$ with a probability of .999.

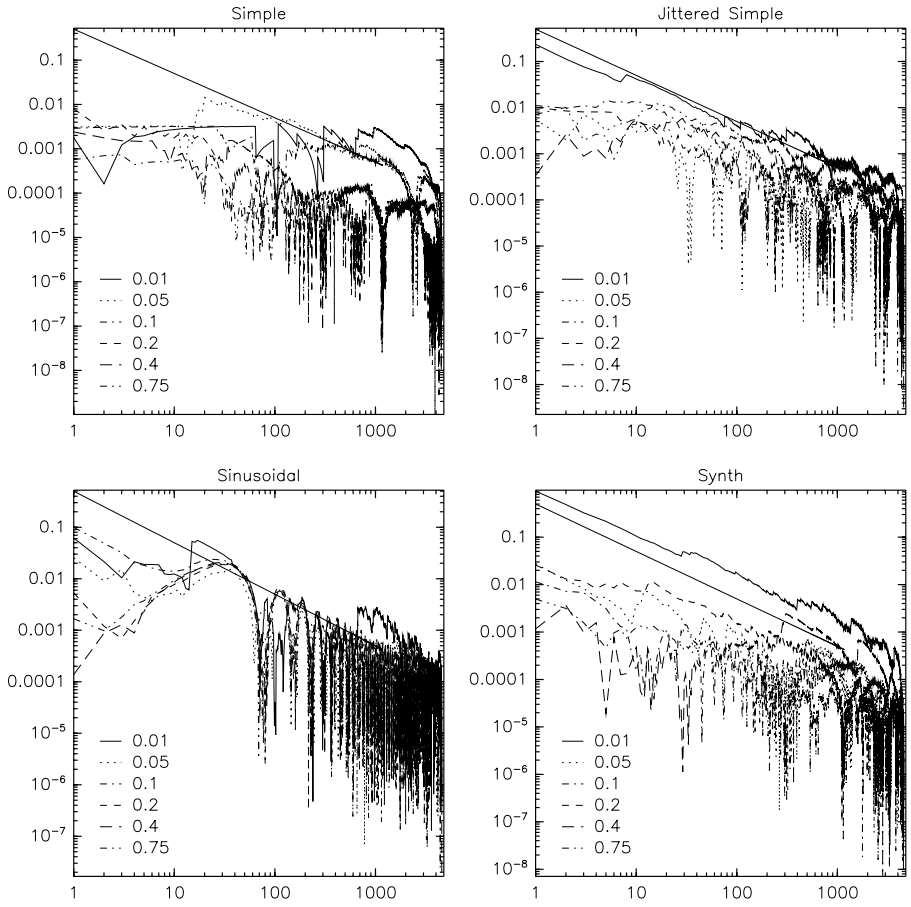

Figure 6.11: The convergence rate of the mean summary statistic: $|m_{p,4000} - m_{p,i}|/m_{p,4000}$ for each basic test performer p. The figure also shows a line corresponding to the line $1/2i$ that bounds all test performers except .01 loading.

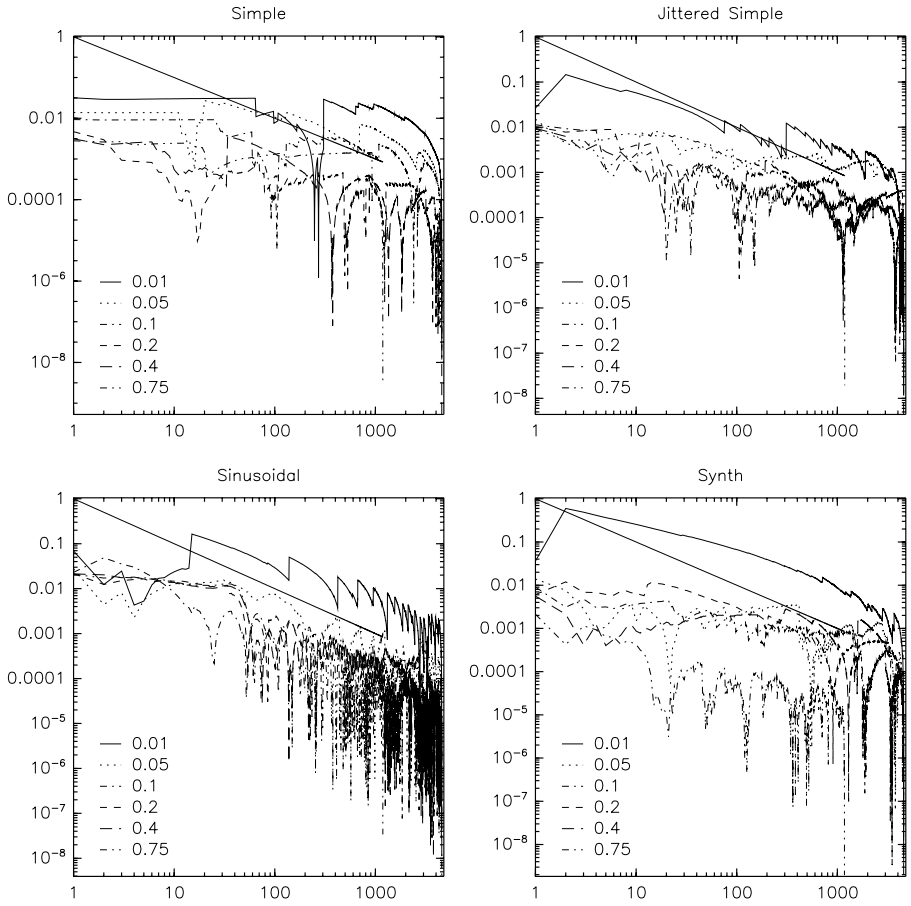

Figure 6.12: The convergence rate of the standard deviation summary statistic: $|s_{p,4000} - s_{p,i}|/m_{p,4000}$ for each test performer i. The figure also shows a line corresponding to the line $1/i$ which bounds all test performers except .01 loading.

at the .01 loading level. Fortunately, inaccurate estimates have the smallest impact on low loading performers.

Per-performer standard deviations also converge sufficiently rapidly for the needs of statistical scheduling. Figure 6.12 shows that, except for the .01 loading test performer, the error in the collected standard deviation $|s_{p,4000} - s_{p,i}|$ is within .01 of $m_{p,i}$ in 10 basic periods and within .001 of $m_{p,i}$ in 100 basic periods. As a result, stat(p) is off by at most $.1m_{p,10}$ after the segregation delay and $.01m_{p,100}$ after the expectation delay. This convergence rate is sufficient for successful statistical scheduling.

As with Figure 6.11, the performers with the largest μ/σ ratios — particularly the .01 loading performer — have the poorest convergence rates in Figure 6.12. Consequently, performers

with large relative standard deviations are disadvantaged twice: first by the slow convergence of $s_{p,i}$ and then by the largest impact of the difference $|\sigma_p - s_{p,i}|$ on their utilization and firmness. However, these are also the performers with the smallest $m_{p,i}$ and so are the ones where inaccuracy least effects the scheduler. As a result, standard deviations for typical performers converge sufficiently within the segregation delay to safely schedule multimedia tasks.

Figure 6.13 shows the convergence rate for the mean of eight entire schedules. The LiquiMedia simulator randomly generated each schedule by selecting a mixture of synthetic test performers with a target loading of 0.80. Table 6.5 lists the performers comprising each of the randomly generated schedules. As with the individual performers, I assumed that the population summary statistics $m_{\mathcal{P},10000}$ and $s_{\mathcal{P},10000}$ for schedule $\mathcal{P}$ approximate the schedule's true mean and variance.

The convergence rate of is $m_{\mathcal{P},i}$ bounded above by $.02/\sqrt{i}$. Approximate mean running times $m_{\mathcal{P},i}$ for the test schedules listed in Table 6.5 converge to within .006 of the full-sequence $m_{\mathcal{P},10000}$ in under 10 basic periods — 130 ms in `liqui`. Further, within 120 basic periods, they lie within .001 of $m_{\mathcal{P},10000}$. This convergence rate permits accurate lifetime admission control decisions within the 200 ms segregation delay of an an audio or video stream.

Figure 6.14 shows the convergence rate for the whole-schedule standard deviation $s_{\mathcal{P},i}$. Values of $s_{\mathcal{P},i}$ are less than $.01m_{\mathcal{P},10000}$ away from $s_{\mathcal{P},10000}$ within 10 basic periods. They are within $.001m_{\mathcal{P},10000}$ within 7000 basic periods. Consequently, stat($\mathcal{P}$) has no more than a .1 error within 10 basic periods. When supported by a working instantaneous admission control mechanism, this convergence rate is sufficient for accurate lifetime admission control.

As with the per-performer results, the whole-schedule results for both the mean and the standard deviation are also "spiky". These y-axis spikes occur because the whole schedule running time random variable $T_{\mathcal{P}}$ is the sum of a number of asymmetric tail-heavy distributions: each new unusually large value of $t_{\mathcal{P},i}$ has an immediate impact on $m_{\mathcal{P},i}$ and $s_{\mathcal{P},i}$ before slowly fading away.

Conductor-collected mean running times for whole schedules comprised of typical performers are within .01 of the true value within the segregation delay of an audio or video stream. Mean running times for individual performers also converge to within .01 of true values within a human's segregation delay. These results show that the conductor generates sufficiently accurate whole schedule or per-performer mean running times within latencies acceptable for admission control. Standard deviation values converge more slowly but, as shown in Section 6.5.3, for typical performers with small σ/μ ratios, the inaccuracies in the standard deviation statistics do not impede successful admission testing.

1	2	3
Jittered Simple 0.1	Sinusoidal 0.2	Synth 0.4
Simple 0.01	Synth 0.05	Sinusoidal 0.2
Simple 0.1	Simple 0.05	Jittered Simple 0.05
Jittered Simple 0.4	Simple 0.4	Synth 0.1
Simple 0.1	Simple 0.05	Simple 0.01
Synth 0.01	Synth 0.05	Simple 0.01
Sinusoidal 0.05		Simple 0.01
Synth 0.01		Simple 0.01
Jittered Simple 0.01		
4	5	6
Synth 0.01	Synth 0.05	Synth 0.2
Jittered Simple 0.1	Sinusoidal 0.05	Synth 0.1
Jittered Simple 0.2	Sinusoidal 0.01	Synth 0.2
Jittered Simple 0.1	Sinusoidal 0.2	Simple 0.05
Sinusoidal 0.2	Sinusoidal 0.01	Synth 0.01
Simple 0.01	Synth 0.1	Synth 0.01
Jittered Simple 0.05	Sinusoidal 0.2	Synth 0.05
Simple 0.01	Simple 0.01	Jittered Simple 0.1
Simple 0.05	Sinusoidal 0.05	Jittered Simple 0.05
Simple 0.01	Sinusoidal 0.1	Synth 0.01
Jittered Simple 0.05	Jittered Simple 0.01	Simple 0.01
7	8	
Jittered Simple 0.05	Jittered Simple 0.4	
Synth 0.4	Synth 0.05	
Simple 0.05	Synth 0.01	
Sinusoidal 0.2	Synth 0.05	
Jittered Simple 0.01	Sinusoidal 0.2	
Simple 0.01	Jittered Simple 0.05	
Sinusoidal 0.01	Synth 0.01	
Jittered Simple 0.01	Simple 0.01	
Synth 0.01	Sinusoidal 0.01	
Synth 0.05	Simple 0.01	

Table 6.5: The table shows the performers comprising each of the random schedules used in the schedule-wide convergence tests shown in figures 6.13 and 6.14.

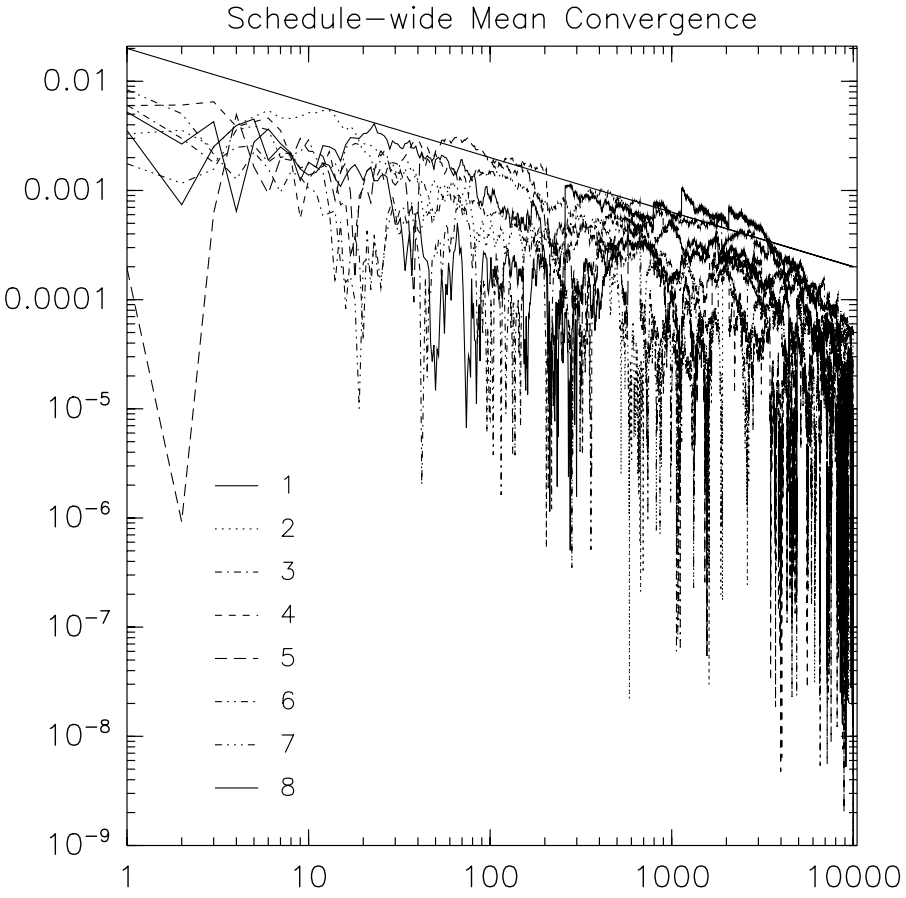

Figure 6.13: Log plot of the convergence of the mean for several entire schedules: $|m_{\mathcal{P},10000} - m_{\mathcal{P},i}|/m_{\mathcal{P},10000}$ and the upper bound $0.02/\sqrt{i}$. The composition of each schedule is shown in table 6.5.

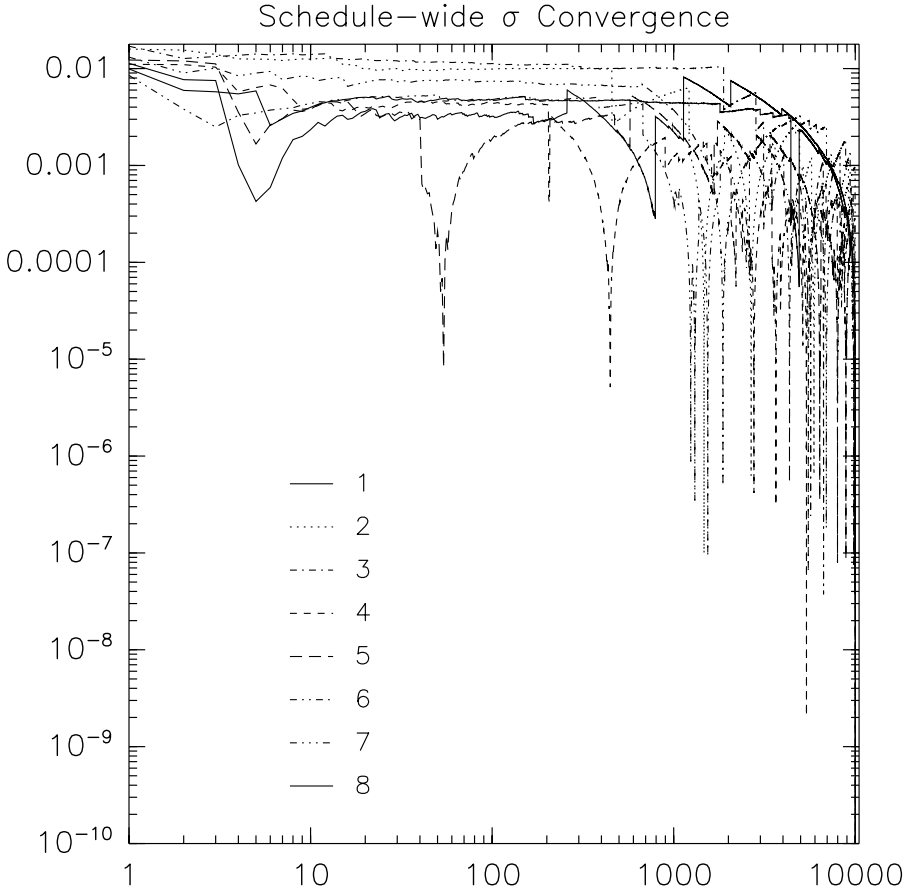

Figure 6.14: Convergence of the standard deviation for several entire schedules: $|s_{\mathcal{P},10000} - s_{\mathcal{P},i}|/m_{\mathcal{P},10000}$. The composition of each schedule is shown in table 6.5.

6.5.2 Instantaneous Admission Control

Once summary statistics are available, the conductor performs an instantaneous admission control test on each performer prior to executing it. It only executes a performer that has at least probability P_c of executing completely before the end of the basic period. Otherwise, the conductor does not execute the performer and thereby causes it to experience a *deferral*.

When a performer attempts to execute past the end of the basic period, it experiences an *overtime*. The conductor suspends overtime performers and removes them from the schedule because LiquiMedia promises to execute performers atomically, The single basic period delay of a deferral is preferable to the possibly many basic periods needed for an application to re-schedule a suspended performer. LiquiMedia's instantaneous admission control mechanism completely eliminates overtimes for a randomly chosen mixture of test performers.

Figure 6.15 compares the fraction of performer executions resulting in overtimes over a range of schedule loading levels with and without the instantaneous admission control mechanism. The graph shows overtime rates from 100 randomly generated schedules, each of which executed in the LiquiMedia simulator for 10000 basic periods.

Figure 6.15 shows that `liqui`'s instantaneous admission control completely eliminates performer overtimes on a randomly-chosen mixture of standard test performers.

6.5.3 Lifetime Admission Control

LiquiMedia's scheduler also has a lifetime admission control mechanism. This mechanism verifies that every path through a candidate schedule has at least P_c of executing successfully. Provided with accurate summary statistics, this mechanism admission tests each path in the schedule using the algorithm presented in Section B.4.5.

The lifetime admission control mechanism operates correctly. Figure 6.16 shows that it correctly rejects every inadmissible schedule from a set of 1662 randomly generated schedules.

The graph is the result of the following experimental procedure. The simulator created 1662 schedules comprised of a random selection of standard test performers. Each schedule contained not less than 400 basic periods of execution. Each schedule's observed P_c was computed by taking the ratio of the number of basic periods in which no overtimes took place over the total number of executed basic periods.

The simulator applied the lifetime admission control test to each of the randomly generated schedules. The lifetime admission control mechanism had a minimum firmness target of 0.9 and performed the admission test using summary statistics from basic period $i = 100$. The randomly generated schedules were binned into a histogram by their observed P_c value. Figure 6.16 shows the proportion of the schedules in each bin that passed the lifetime admission

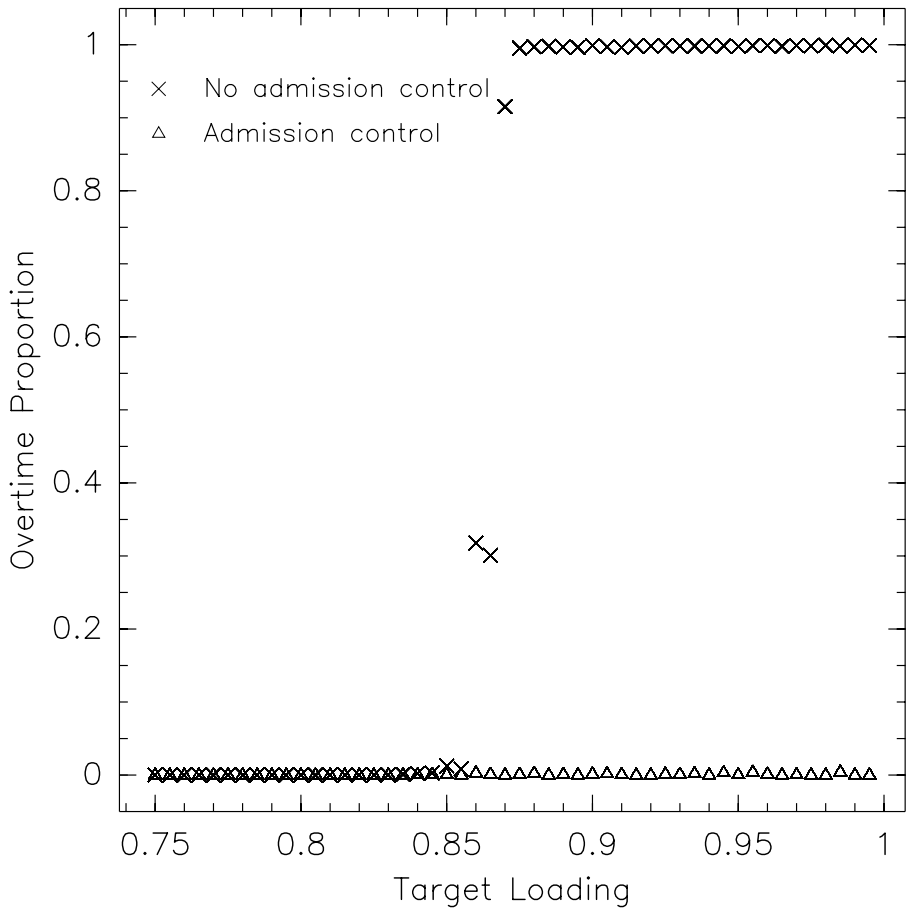

Figure 6.15: The graph compares the normalized number of overtimes as a function of a desired schedule loading level with the instantaneous admission control mechanism (triangles) and without it ("x"). Instantaneous admission control completely eliminates overtimes.

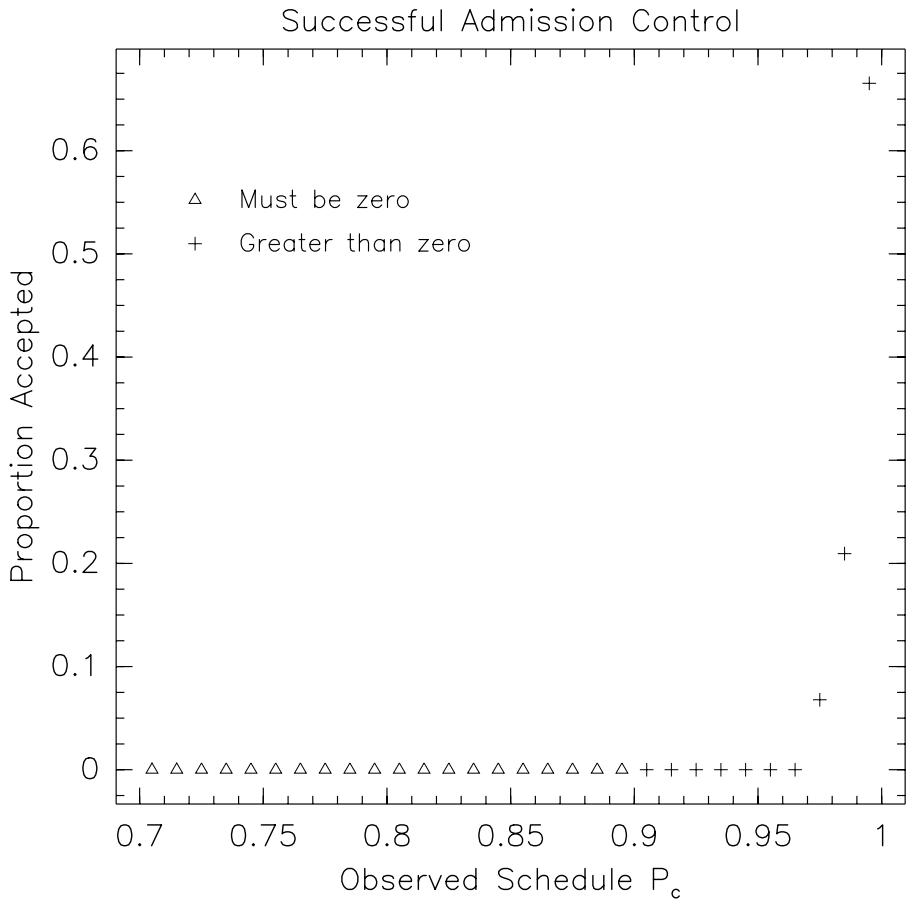

Figure 6.16: The graph shows the proportion of schedules admitted at a target firmness of $P_c = .9$, using summary statistics computed at $i = 100$ over a domain of the schedules' observed fraction of basic periods in which no overtimes occurred. The graph shows that the lifetime admission control mechanism operates correctly: no schedule with firmness less than .9 (triangle) is wrongly admitted.

control test. The figure shows that the admission test operates correctly as it does not wrongly admit any schedules (plotted with triangles) whose observed firmness is less than .9.

Figure 6.16 also suggests that LiquiMedia's admission control mechanism might be pessimistic. While every admitted schedule has at least 0.9 probability of executing without overtimes, some of the rejected schedules may have more than 0.9 probability of executing successfully.

Experimental limitations explain some of the observed pessimism. The success or failure of a schedule in each basic period is an independent binary random variable. The domain of Figure 6.16 is the proportion of basic periods in which a schedule actually executed successfully. This number is normally distributed around the schedule's actual but unknown probability of success. Highly variable schedules simulated for only a short number of basic periods can execute successfully in every basic period but still have a finite probability of failure.

The admission control mechanism's over-estimate of the schedule's standard deviation at basic period 100 explains the remainder of the pessimistic rejections. The Chebyshev-style estimator computes a pessimistic bound for a schedule $\mathcal{P}$ when $s_{\mathcal{P},100}$ considerably exceeds $\sigma_{\mathcal{P}}$. The admission control mechanism incorrectly rejects some schedules on the basis of this pessimistic bound.

LiquiMedia's feedback mechanism significantly reduces the impact of this pessimism in practice. In normal operation, the scheduler adds only a single new application to an existing schedule comprised of established applications. The performers comprising the established applications typically have summary statistics spanning many thousands of basic periods and so, as shown in Figure 6.14, have accurate summary statistics. This majority offsets the inaccurate standard deviations of performers newly added to the candidate schedule and enables the scheduler to admit a wider range of highly loaded schedules.

This section's results demonstrate that LiquiMedia's lifetime admission control mechanism always detects a schedule with less than firmness P_c. Consequently, this mechanism provides the decision functionality needed to satisfy the principle of modularity.

6.5.4 Feedback and Schedule Convergence

Satisfying the principle of modularity requires that LiquiMedia support the execution of independently-authored performers. Doing so has three architectural requirements. First, as shown in Section 6.5.1, LiquiMedia can *measure* the running time of independently-authored performers and schedules. Second, as shown in Sections 6.5.2 and 6.5.3, LiquiMedia must be able to use these measurements to *decide* if individual performers or whole schedules are admissible. Finally, LiquiMedia must be able to *correct* a schedule that was previously correct but is no longer so.

LiquiMedia's schedule feedback mechanism satisfies this final requirement of the principle of modularity by design. LiquiMedia's feedback mechanism consists of the conductor providing updated statistical summary information to the scheduler. The scheduler then verifies that the current schedule still has at least P_c of completing in a single basic period.

So long as accurate per-performer and schedule-wide statistical summary information is available and the admission control test operates as designed, the scheduler can always successfully convert an incorrect schedule into a correct schedule. LiquiMedia corrects an incorrect schedule by removing performers from it in order of increasing importance until the schedule is correct. Because the feedback is negative, the schedule is guaranteed to converge to a correct schedule.

As discussed above in Section 6.5.3, the conductor will gather accurate summary statistics for performers within 50 basic periods and accurate means for whole schedules within 10 basic periods. The scheduler re-verifies the current schedule after 10, 35, 105, 4561 and then every 9123 basic periods thereafter. These values correspond approximately to .1, .5, 1.5, 60 and 120 seconds.

Three criteria guided the selection of these times: the perceptual thresholds discussed in Section 1.1, the convergence times of the summary statistics discussed in Section 6.5.1 and the need to prevent numerical overflow in conductor-collected integer statistics as discussed in Section B.3.6.

Schedule verifications after 10 basic periods eliminate inadmissible performers prior to the 200 ms segregation delay of a human's auditory or visual modality. The verification at .5 seconds improves the per-performer standard deviations available to the instantaneous admission control mechanism. The verification at 1.5 seconds tests the schedule before a human's expectation threshold and after the availability of more accurate per-performer and whole-schedule statistics. The test at one minute verifies the schedule once accurate standard deviations are available. Finally, re-verifications every two minutes keep accumulated sum-of-squares values from overflowing.

The chosen reverification times also provide sufficient time for the scheduler's latency. For schedules with under ten performers and realtime loadings less than .75, the scheduler can verify a schedule between successive conductor invocations. When the schedule contains upward of a 100 performers, scheduler latency is under 8 basic periods from scheduling request to posting.[8]

The performers responsible for the generation of well-established realtime streams may change their behaviour. Perhaps through a coding error, data alteration or deliberate attack on the system, a mature performer may suddenly start to execute for a considerably longer or more variable amount of time than the execution profile on whose basis it was admitted into the schedule.

[8]Schedule life cycle is explained in Section B.4.6.

The scheduler successfully handles this "embedded leak" situation by design. Unfortunately, the scheduler's negative feedback heuristic cannot handle this situation without disrupting the operation of other possibly established streams. The scheduler always tries to correct a schedule by removing the least important scheduled performers. If an embedded leak is merely symptomatic of a performer requiring additional processor resources to respond to user requests, then the negative feedback mechanism will correct the situation within a minute or, if the embedded leak has caused an overtime, within 1 to 8 basic periods.

Unfortunately, the negative feedback mechanism's response suspends many correct performers when an important performer causes the embedded leak deliberately or through programmer error. However, it is not possible to automatically determine if a performer's use of additional processor resource constitutes a programming defect or a genuine need.

6.6 Summary

This chapter has presented a number of performance tests on the `liqui` prototype and simulator whose results confirm that LiquiMedia's design satisfies the four principles of a multimedia operating system. The tests measure the limitations in the `liqui` prototype and show that it operates successfully as designed on a range of typical performers. In particular, the chapter has shown the following.

- The audio player tests show that `liqui` satisfies the principle of partitioning.
- `liqui` implements the principle of synchronous realtime by design. Its use of the Chebyshev estimator delivers equal or better expected utilization than the WCET estimator for non-pathological test performers and vastly superior expected utilization levels for realistic audio-processing performers.
- `liqui` satisfies the principle of ultra-fine granularity because its performer implementation is 9.48 times more efficient than the thread task abstraction on the same machine.
- `liqui` can measure, admission test and correct schedules comprised of arbitrary mixtures of typical performers within a human user's 200 ms segregation threshold and obtain accurate per-performer summary statistics within a human user's 1.5 second expectation threshold. This capability satisfies the principle of modularity.

Chapter 7

Future Work

LiquiMedia has been a large project. Consequently, this thesis defers many interesting areas of investigation. This chapter describes some of these areas. It begins with improvements to the LiquiMedia implementation and experiments. Then the chapter presents more conceptual extensions to the research and finally product versions of the LiquiMedia architecture. Within each category, possible future work is presented in order of increasing ambition.

7.1 Experimentation and Analysis

Chapter 6 describes an empirical validation of LiquiMedia's operation on a range of typical cases. This section discusses experiments that widen the scope of the empirical validation and explore more of the prototype's performance envelope.

7.1.1 Scheduler Operation

LiquiMedia's scheduler does not operates realtime. This design aspect reduces overhead inside the realtime code path and therefore helps LiquiMedia satisfy the principle of ultra-fine granularity. However, because it does not operate in real-time, no experiments evaluated the scheduler's throughput and running time.

The scheduler's running time overhead is proportional to the number of schedule paths and performers. LiquiMedia cannot support schedules that take longer to admission test than the time available between the most frequent schedule re-verifications. Consequently, the scheduler's overhead bounds the maximum possible number of performers. Future research will include deriving theoretical limits on the time needed to verify a schedule and the maximum number of performers supported at any given re-verification interval.

This theoretical derivation also requires corresponding experiments. It is important to measure the following:

(i) the time needed to update the summary statistics as a function of the number of performers,

(ii) the time needed to admit a schedule as a function of the number of performers and paths through the schedule graph, and

(iii) the average latency of a schedule re-verification from the moment when the conductor requests one until the scheduler posts the new re-verified schedule.

These results determine the processor reserve needed to guarantee adequate schedule verification rates. The following approaches can lower this overhead if it proves excessive.

(i) Reduce the cost of schedule verification by caching path fragments from previous verification attempts.

(ii) Modify the conductor to perform usage-tracking on schedule paths and to re-verify only those paths for which performers have generated new statistical summary information.

(iii) Explore the impact of reducing the schedule verification rate.

7.1.2 Feedback Tests

The scheduler's negative feedback mechanism is guaranteed to reduce an incorrect schedule to a correct schedule. However, the existing results do not include a measure of how long it takes the scheduler to remove incorrect performers from the schedule. Consequently, it is important to add simulation of schedule feedback to the LiquiMedia simulator. Subsequent research will then explore the operation of the scheduler as it handles incorrect performers and measure the convergence rate from an incorrect to a correct schedule.

The envisaged experiment performs admission control tests with both the simulator and `liqui` on randomly generated schedules at a variety of schedule ages and records the evolution of the schedules. The incorrect schedule consists of a mixture of correct and incorrect performers where incorrect test performers are constructed from the existing test performers using the three following techniques.

Enlarge Enlarge the mean running time of one of the performers described in Section 6.1.3 so that its estimated running time exceeds the available time.

Vary Increase the variance of an instance of Sinusoid or Jittered so that its mean running time is less than the available running time but its estimated running time stat(p) exceeds the available time.

Grow After some configurable number of basic periods, grow the performer's mean or variance until its estimated running time can no longer be accommodated. The slope of the increase is configurable at the time of performer creation and includes the important special case of a step function.

The results include the number of basic periods that `liqui` spends on the schedule life cycle, the average depth of the verification request conduit and the correlation between schedule verification rates and the operation of the instantaneous admission control mechanism.

7.1.3 Varied Loadings

The LiquiMedia simulator uses only statistical profile data generated by the `liqui` prototype with the typical test performers described in Section 6.1.3. However, future development will include extending the simulator to synthesize statistical profile data corresponding to both realistic multimedia loads and performers with extreme σ/μ ratios. The following running time distributions will be implemented:

(i) the Pareto distribution shown in Figure 6.2(1),

(ii) the "negative" Pareto shown in Figure 6.2(2),

(iii) per-frame MPEG decompression timings, and

(iv) frame-drawing timings in an interactive video game such as Quake.

The experiments on the scheduler described in Chapter 6 can then be repeated with these additional performer types. In particular, this work will further demonstrate that the Chebyshev-style estimator maintains its utilization advantages over the WCET estimator for a more diverse selection of performer loadings.

7.1.4 Expanding the Envelope

The simulations and experiments described in Chapter 6 do not fully explore LiquiMedia's performance envelope. Many interesting issues remain. Future research will exercise LiquiMedia over a wider range of firmness and loadings.

High load experiments have LiquiMedia running with an average load in excess of an entire basic period. By deactivating the lifetime admission control mechanism, a modification of the peak loading test can force conductor loads that exceed 1.0. Thanks to the instantaneous admission control mechanism, such a configuration should retain the configured performer

firmness while possibly delivering increased utilization levels. It will also further validate the operation of instantaneous admission control mechanism.

Second, additional research will include conducting experiments over a wider range of performer firmness values. The existing results are only for firmness $P_c = .99$ and .95. These values are appropriate for multimedia stream generation applications. However, tests at more conservative and more aggressive settings will evaluate LiquiMedia's suitability outside of multimedia stream generation applications.

7.1.5 Comparisons

This thesis has only compared LiquiMedia to RMS scheduling. The results confirm that under conditions appropriate for multimedia applications, statistical scheduling provides, at the cost of hard realtime guarantees, superior whole-schedule processor utilization compared to RMS. The literature contains a wide variety of other scheduling techniques that also sacrifice hard realtime guarantees to improve processor utilization. The results of future research should compare LiquiMedia's utilization, performance and handling of untrustworthy performers to operating systems that use rate-based scheduling such as SMART and BERT [Nie99, BPM99].

7.1.6 Time Series Analysis

The conductor collects per-performer and per-schedule statistical profile data. This information is a time-series. There is an extensive mathematical literature concerning time-series analysis [FBL+03]. Incorporating results from this literature into LiquiMedia could improve the accuracy of the scheduler.

7.2 Prototype Enhancements

I bounded `liqui`'s development time by deferring a variety of interesting but non-essential features. Completing some of these features can significantly increase `liqui`'s performance. Others are natural generalizations that warrant investigation. This section describes such extensions to the prototype.

7.2.1 Reducing Overhead

Two implementation optimizations will further reduce `liqui`'s already low per-performer overhead. First, the LiquiMedia prototype's handling of saved stack frames for trapping per-performer exceptions should be rewritten in assembly language. The existing implementation

uses `setjmp`. This call saves more state than necessary. An assembly language implementation would save the minimum amount of processor state. Second, the conductor's computation of summary statistics also needs to be re-written in assembly language. In particular, a by-hand implementation of the computation of $s^2_{p,i}$ and $m_{p,i}$ should take less than ten assembly instructions.

7.2.2 Timing

Conversion of the prototype to use hardware timing promises to have an even greater impact on LiquiMedia's per-performer overhead. The prototype used the `gethrtime` function call to obtain monotonically increasing high-precision time values. Figure 1 shows that `liqui` invokes this routine at least twice for every performer executed by the conductor. Both the accuracy of the timing information returned by this routine and the overhead of the conductor itself are limited by the running time of the `gethrtime` function.

As discussed in Section 6.4.2, the `gethrtime` function call takes approximately 545ns to complete. Future work will therefore include replacing the calls to the `gethrtime` function with an assembly language macro that provides equivalent functionality via the cycle-counting facilities in the UltraSparc processor.[1]

Switching the `liqui` prototype to hardware-assisted timing should reduce the per-performer overhead of the conductor by at least 70%. Repeating the peak-loading test described in Section 6.1.4 will measure the extent of the improvement.

7.2.3 Enhanced Conduits

While not a direct contributor to per-performer overhead, the implementation of `liqui`'s conduit library can also be improved. The current conduit implementation uses the Solaris kernel's mutual exclusion primitives and therefore exposes conduit-using performers to kernel call and kernel scheduling overhead.

A more efficient implementation should be possible. Conduit primitives can use the atomic swap assembly language instructions found on recent Sparc and IA32 hardware [Int99, SPA93]. A careful combination of atomic swap instructions with a well-designed spin-lock permits implementing the conduit read and write operations in fewer than ten assembly language instructions.

[1] Processors such as the UltraSparc and IA32 have efficient hardware cycle counting facilities [Int99, SPA93].

7.2.4 Per-Performer Firmness

LiquiMedia provides a flexible trade-off between utilization and firmness. The `liqui` prototype does so globally with the P_c parameter. As a result, every performer shares the same firmness/utilization trade-off. This restriction is arbitrary and can be eliminated. A future version of `liqui` will provide per-performer firmness settings.

The LiquiMedia prototype provides a mechanism for adjusting an application's importance. Per-performer firmness will be integrated into this mechanism so it permits independent adjustment of firmness and importance.[2]

How to use independent firmness and importance remains a subject of future research. One particularly interesting open issue is the user interface for a graphical tool that permits the user to adjust the relative importance and firmness of applications in a way that maximizes user satisfaction. The programmatic API for adjusting both firmness and importance will also require investigation.

7.2.5 Instrumentation

Lastly, the `liqui` prototype can benefit from improved instrumentation. Modifying the conductor to record complete whole-schedule statistical summary data including the path taken through the schedule graph will provide additional valuable experimental data. Also, the scheduler will record the running time of schedule verifications. An extended instrumentation library will integrate this new data with the existing facilities for recording per-performer and per-schedule running time information.

Several miscellaneous improvements are also desirable. The `liqui` prototype will label data sets in a standardized way and insure that each dataset includes a representation of the schedule graph. The `liqui` prototype should also successfully preserve as much accumulated data as possible in the advent of a crash.

These enhanced data recording facilities lay the foundation for a general persistence mechanism in LiquiMedia. Accumulated statistics will be preserved between executions to permit restarting performers with previously accumulated summary statistics.

7.2.6 Processor Redistribution

LiquiMedia preserves established applications at the expense of new applications. This is a desirable default behaviour. For example, starting a new application should never jeopardize

[2]This independence has practical limitations. Some combinations of firmness and importance significantly increase the likelihood that a performer will not be scheduled.

the operation of an existing source of a multimedia stream. However, its rigidity can also be a weakness. For example, the user might wish to reduce the processor consumed by a video player so that a video conference can deliver a higher quality of service. Consequently, LiquiMedia needs a mechanism that permits applications to contend for recently freed processor resources.

The `liqui` prototype has a partial implementation of such a mechanism. The implementation of the processor redistribution mechanism should be completed and tested to insure that the scheduler's feedback mechanism remains stable while applications contend for available processor resources.

The user interface to this API also requires research. A UI for the processor redistribution mechanism must clearly show that the amount of processor is finite and increasing one application's allocation is always at the expense of the other applications.

7.3 Multiprocessor Support

At present, `liqui` permits only a single conductor. While non-realtime threads can freely execute on the remaining processors, this arrangement does not take maximum advantage of multiprocessors. However, the existing structure does handle the important special case of a general purpose processor scheduling performers for a single conductor running on a separate purpose-built multimedia processor.

7.3.1 Multiple Conductors

A more general arrangement permits one conductor instance for each general purpose processor in a symmetric multiprocessor machine. Successfully implementing such an extension while still satisfying the principle of modularity requires additional research in two areas.

First, the existing conduit mechanism permits performers to access conduits cheaply because the conductor maintains a global lock on conduit access. Two conductors cannot successfully execute at the same time if they must contend for the same global conduit lock. A multiprocessing LiquiMedia implementation requires a conduit implementation that permits two different conductors to finely interleave access to a single conduit. Resolving this issue is primarily an implementation challenge.

Second, conduits provide no mechanism for a single application to synchronize the execution of performers running on different processors except at basic period boundaries. The resultant latencies for inter-performer communication are impractical. Further, the principle of synchronous realtime permits developers to assume that no two performers ever execute simultaneously. Performers executing in separate conductors violate this assumption.

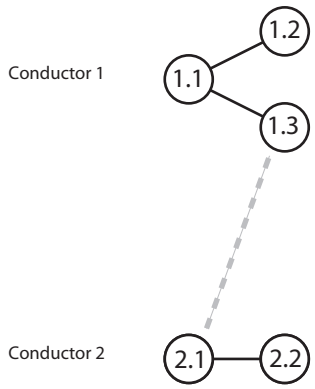

Figure 7.1: The dotted edge is a probabilistic order dependency. It indicates that 1.3 must execute after 2.1 even though $1.3 \in \text{Adj}(2.1)$.

The most general solution permits multiple conductors to simultaneously take different paths through a unified schedule graph while respecting an application's constraints on parallelism. Consequently, future research will generalize the notion of a schedule graph so that an application can specify order constraints on performer execution.

Applications specify these constraints with special edges that I call *probabilistic order dependencies*. A probabilistic order dependency indicates that the target performer must execute after the source performer but does not have to be in the source performer's adjacency set.[3] Figure 7.1 shows an excerpt from a schedule graph where performer 1.3 requires that both performer 2.1 and 1.1 have completed before it can execute but performers 1.1 and 2.1 may execute simultaneously.

Probabilistic order dependencies expose inter-conductor performer synchronization to analysis by the scheduler. Data safety is guaranteed by insuring that the performers communicate only through conduits. However, for the application to be admitted to the schedule, performer 1.3 must be positioned in the schedule such that both 1.1 and 2.1 have at least the application's specified firmness of completing before 1.3 is invoked. An application would declare the dependency between these three performers at the time of first submitting its candidate application schedule graph and the scheduler would attempt to satisfy the order constraints.

This scheduler will be considerably more complicated than LiquiMedia's current scheduler because it can no longer treat the execution time of each performer as an independent random variable. Instead, a performer's probability of executing correctly depends on the tree of its order dependencies. The scheduler uses these constraints to compute an earliest execution time in the basic period to complement the existing latest execution time.

[3]Section B.4.1 describes LiquiMedia schedule graphs in detail.

7.3.2 NUMA

An increasing number of multiprocessor systems are NUMA machines. Even desktop systems and game consoles are built with NUMA architectures [Adv03, Spo03]. A symmetric multiprocessor version of LiquiMedia on a NUMA architecture faces an additional complication: the running time of a performer can vary substantially depending on which processor's local memory hosts its code and data.

LiquiMedia's statistical scheduling should transparently handle the additional running time variance of NUMA when the difference between local and remote memory accesses is small. Future empirical research is needed to quantify "small". However systems such as a loosely connected cluster where the level of non-uniformity is not "small" will require extensive modifications to LiquiMedia's scheduling mechanisms.

7.4 DAG Scheduling

LiquiMedia's schedule graph is a DAG where each node corresponds to a performer. A compiler constructs a similar data structure while generating code where each node is a basic block with a fixed running time. This structural similarity suggests that a compiler can generate code in the form of a schedule graph of performers and so automatically convert traditional threaded code into a LiquiMedia application.

However, a compiler-created schedule graph where each basic block corresponds to a performer will be enormous. Handling such large graphs poses open research challenges. Graph compression based on storing pointers as variable length bit vectors instead of full machine words can reduce the size of the graph but is insufficient for any non-trivial application.

Instead, path pruning techniques such as discarding probabilistically unlikely paths warrant investigation. If the implementation can successfully operate on large schedule graphs, it could *automatically* divide applications into many ultra-fine grain performers. Further, statistical profile profile information for such a schedule graph provide a rich source of branch prediction information. This technique could be particularly useful in the implementation of a JIT compiler.

7.5 Bandwidth Allocation

This thesis has only considered statistical scheduling for the allocation of processor resources. A sizable literature already describes the use of statistical models for call attempt admission

control to cellular networks and the resultant allocation of network bandwidth.[4] Future research will investigate which, if any, of these results are applicable to processor scheduling and, if any are, will use them to improve LiquiMedia.

A more interesting future research topic is to investigate the use of statistical models of memory and I/O bandwidth usage for performer scheduling. Under this generalization, a performer has the existing statistical model of its running time. However, it also has a statistical model of its I/O and main memory bandwidth needs. The admission control mechanism would consider both I/O and processor needs before accepting a performer into the schedule.

Combining this extension with the DAG scheduling extension discussed in Section 7.4 can provide some interesting features. In particular, the scheduler can attempt to overlap the execution of I/O and processor-bound performers to further improve system utilization. Testing the success of this generalization involves implementing and empirically validating the operation of applications that require a mixture of I/O-bound and processor-bound tasks such as realtime databases and video-on-demand servers.

7.6 Native Implementation

The previous sections have described various improvements to LiquiMedia. Despite that `liqui` is built on top of Solaris, it remains possible to extend `liqui` with most of the small-scale improvements described above in Section 7.2. However, more sophisticated enhancements such as support for multiple conductors require an implementation of LiquiMedia running natively on the hardware because Solaris does not provide the necessary primitives.

Extending the Linux kernel is one way to build a native LiquiMedia implementation without needing to write an entirely new operating system. An alternative is to extend a Java virtual machine to run natively on the hardware.

Only a native implementation can provide complete control over interrupts and full access to the hardware. Such control is a prerequisite of features such as scheduling based on bandwidth needs or support for multiple conductors. Complete control of interrupt execution also permits accounting for their running time separately from that of performers.

A new native implementation requires repeating all of the tests discussed in Chapter 6 and above in Section 7.1. In particular, future research should compare the implementation cost and complexity of tracking performer running times excluding interrupts versus the simpler existing approach.

[4] The search string "call admission" at [NEC03] generates upward of a thousand paper matches. Capone and Stavrakakis's paper provides a (randomly chosen) example: the paper describes a statistical model for call admission in TDMA networks [CS98].

7.7 Realtime Java VM

As described further in appendix D, I partially implemented a realtime Java virtual machine based on the LiquiMedia prototype. Completing this JVM requires additional research to address the large running time variances caused by garbage collection, the VM's dynamic tear-down re-compilation and its dependencies on the host operating system for media delivery.

7.7.1 Garbage Collection

Java code makes frequent (perhaps excessive) use of the heap. A complete realtime VM must permit memory allocation by realtime code. These allocations must also complete in constant time. Oberon's memory management strategy suggests one possible solution [Wir92]. By keeping the performer heap inaccessible to all non-realtime code, all performer allocations persist for only the duration of the basic period. This restriction trivializes realtime garbage collection to an $O(1)$ pointer manipulation.

This solution eliminates a difficult problem by taking advantage of LiquiMedia's architecture. However, it does pose the problem of how to support the execution of typical Java programming idioms that require reference passing between performers and non-realtime code. Future research should investigate mechanisms including copying objects through the conduit mechanism, using conduits for updates to shared objects and enhancements to the Java verifier to enforce any necessary restrictions on Java performers.

Implementing a single realtime garbage collection mechanism for the entire JVM is a more traditional alternative. The literature contains extensive discussion of realtime garbage collection techniques and does not contain any one obviously superior technique [Wil96]. This suggests that developing a good realtime garbage collection mechanism is a difficult and perhaps even impossible endeavour.

7.7.2 Tear-Down

Java is a realtime-hostile environment. The VM may choose to interpret a bytecode method or run a compiled version instead of depending on the circumstances. In particular, the VM implements a concept called "tear-down" where a previously compiled method is discarded because loading a different class alters the compiled method's use of virtual methods [PVC01]. Tear-down can radically alter the running time of a performer long after it has been scheduled and consequently cause undesirable scheduling failures.

LiquiMedia supports performers that rapidly raise their running times using the schedule restructuring interfaces described in Section B.4.4. Future research will verify the extent to which these successfully handle the running time variance caused by the VM's tear-down and

recompilation behaviour. Alternatively, VM architectures without "tear-down" might prove a better fit for LiquiMedia [GH01]. Future research should also touch on this issue.

7.7.3 I/O

Finally, most existing Java virtual machines are hosted by an existing operating system and therefore lack efficient and low-latency access to multimedia stream generation hardware such as graphics and sound processors. As a result, they cannot successfully generate realtime streams. Rectifying this problem requires that either the host operating system provide LiquiMedia-style realtime primitives (a JVM running as a partitioned child operating system for example) or the realtime JVM executes natively. Both solutions require extensive implementation.

Java is a safe programming language. Authoring performers in Java permits independently authored performers to share a single memory space. This simultaneously helps satisfy the principle of modularity and the principle of ultra-fine granularity realtime. Should a realtime Java VM prove pointless or overly difficult, other solutions for software-enforced memory safety require investigation [JMG+02].

7.8 Generative Operating Systems

General purpose operating systems provide mediocre service to all application domains. LiquiMedia better supports multimedia applications because it is specialized for their needs. LiquiMedia represents one particular instance of a domain-specialized operating system. Other application domains exist that could benefit from an operating system specialized for their specific needs.

Simply generalizing LiquiMedia eliminates the very advantages provided by specialization. Work on generative programming suggests an interesting alternative [CE00]. Rather than implementing a general purpose operating system, implement an operating system *generator*. The generator constructs an entire family of optimized operating systems by combining standardized components. Component selection is driven by architectural principles of the intended application domain. Each component implements a design aspect such as those described in Chapter 2.[5]

Future research would include the development of a dynamically reconfigurable generative operating system. Such an OS can reconfigure itself at runtime to any one of the specialized operating systems that the generator can produce. For example, dynamic generative OS would

[5]Additional design aspects would be needed to support operating systems specialized for non-realtime application domains.

adopt a LiquiMedia-like architecture when running a mixture of multimedia tasks but would have an traditional RMS architecture when running process control applications with differing periods.

7.9 Summary

Many other improvements and extensions exist. Near-term effort will correct oversights in the prototype and improve the experiments and analysis. Longer-term research will explore the more difficult and rewarding issues of multiprocessor support, DAG scheduling's integration of compiler and scheduler and its possible implementation inside a realtime JVM, scheduling by bandwidth allocation, and a generative operating system framework.

Chapter 8

Conclusions

This thesis has presented the LiquiMedia operating system architecture. LiquiMedia is a *multimedia operating system* — an operating system architecture designed to allocate processor resources to multimedia applications. Multimedia applications generate information streams for human observers.

The information stream is a concept of perceptual psychology. Humans pre-attentively segregate time-varying data from all their senses into a number of information streams within 200 milliseconds [Bre90]. Information streams such as orchestral music nest recursively: the orchestra has strings and woodwinds, the woodwinds have flutes and clarinets. A single stream can consist of information perceived by different sensory modalities: a video has both sound and image streams.

For the mind to successfully segregate an information stream, it must exhibit continuity, synchronization to within 20 milliseconds [HS61] and exceed a minimum threshold duration of approximately 200 milliseconds [Bre90, OR86]. Once segregated, a stream tolerates occasional brief interruptions. LiquiMedia allocates resources to stream-generating applications. Consequently, it insures that stream-generation tasks exhibit continuity and easily synchronize while remaining sufficiently free from interruption.

LiquiMedia takes advantage of the properties of human stream perception to increase efficiency and simplify application development. A human tolerates occasional interruptions in an information stream. Consequently, LiquiMedia increases operational efficiency by sacrificing deterministic realtime guarantees.

A stream has a 200 millisecond segregation delay. LiquiMedia determines the running time of media-generation fragments empirically during this delay. This way, the operating system neither requires nor relies on developer-provided application running time data so LiquiMedia can safely execute independently-authored tasks. Safe execution of independently-authored

tasks permits sub-dividing media-generation applications into simple re-usable components. Further, LiquiMedia eliminates error-prone thread synchronization code by invoking tasks in a synchronous framework. These features simplify application development.

8.1 Design Principles

It was not by accident that LiquiMedia provides simpler development and efficiency advantages. I first defined architectural principles drawn from the properties of information streams and multimedia application development. These principles then guided the design.

I defined four principles: processor partitioning, modularity, synchronous realtime and ultra-fine grain realtime. Processor partitioning requires explicit separation between realtime and non-realtime computation in the implementation and the developer's mind. Modularity requires execution of multiple independently-authored realtime tasks. Synchronous realtime requires reliable realtime execution where all tasks sharing a single external clock. Finally, the ultra-fine grain principle requires efficient support of large numbers of realtime tasks.

The principles of partitioning and synchronous realtime simplify application development by eliminating priority inversions and the challenges of mutual exclusion. The principle of modularity puts the *multi* into LiquiMedia's suitability for multimedia applications by supporting many separate applications generating many different streams.

The principle of ultra-fine grain realtime demands efficient execution even of finely-divided object-oriented software. Finally, the principle of synchronous realtime insures that realtime tasks easily satisfy both the continuity and inter-stream synchronization requirements of a human audience.

I classified existing RTOS designs into a taxonomy organized by their common design aspects. The combinations of design aspects shown by the previous RTOS approaches do not satisfy the architectural principles. The taxonomy suggested several otherwise untried combinations of design aspects. One combination promised to satisfy the architectural principles. It formed the basis of LiquiMedia's design.

Having chosen principles whose implementation would result in a multimedia operating better able to support the needs of stream-generation applications, I designed LiquiMedia to implement these principles.

8.2 Design Aspects

LiquiMedia has the following design aspects: hierarchical partitioning, the performer task abstraction, distributed scheduling, statistical admission control, impermeable hierarchical par-

titioning, empirical statistical admission control, soft realtime deadline sensitivity and asynchronous message passing IPC. This combination promised to satisfy the four principles that I require for a multimedia operating system. I implemented a LiquiMedia prototype and verified empirically that this selection of design aspects cooperates to satisfy the architectural principles.

LiquiMedia uses hierarchical partitioning to provide two child operating systems: the conductor RTOS and the non-realtime composer operating system. Tasks hosted by these two child operating systems communicate via asynchronous message passing. The conductor admission tests and, if successful, invokes performer-style tasks from an aggregate of per-application schedules. It also collects per-performer statistical profile data in this aggregate. A non-realtime scheduler task constructs ever more refined schedules at the request of applications or the conductor.

LiquiMedia uses the performer-style task abstraction. Performers are the most efficient task abstraction because they execute without preemption and can task switch with a single assembly language jump statement. However, an operating system must have an admission control mechanism to ensure that atomic performers have an adequate probability of completing voluntarily before their deadlines.

An admission control mechanism imposes overhead. Despite this overhead, the LiquiMedia prototype's performers are 9.48 times more efficient than Solaris threads that do not have any admission control overhead. The combination of admission control and the performer-style task abstraction has a clear efficiency advantage over threaded task abstractions while providing added functionality.

LiquiMedia's realtime conductor spends part of the per-performer overhead collecting performer profile information. LiquiMedia's scheduler computes a performer's mean and standard deviation from this profile information and uses these statistics for admission control.

Schedule feedback limits the lifespan of any particular schedule and therefore bounds the amount of accumulated statistical profile data. This permits the conductor to collect per-performer profile data in at most 64 bits of precision using only integer arithmetic. The scheduler computes numerically accurate means and standard deviations from the accumulated per-schedule data with the pair-wise algorithm.

More importantly, the collected per-schedule statistics converge to values sufficient for admission control within the user's 200ms stream segregation threshold. LiquiMedia schedules new performers without requiring correct estimates of their running time. Instead, it executes, profiles and accepts or rejects new performers before the user has successfully segregated any stream that they might have attempted to generate.

The scheduler's estimated mean running time of a number of randomly generated schedules converged to within 1% of the full sequences' means within 10 basic periods. Given that the

prototype has a 13ms basic period, LiquiMedia's scheduler can accept or reject performers within 130ms.

This property permits LiquiMedia to satisfy the principle of modularity. LiquiMedia's scheduler in no way depends on applications to provide estimates of a performer's running time. Instead, it collects enough information to safely schedule or reject new performers within 200 milliseconds.

The LiquiMedia conductor not only collects per-performer profile information. It implements an instantaneous admission control mechanism to satisfy the principle of partitioning and synchronous execution. Before invoking any performer, the conductor verifies that the performer has at least a specified minimum probability of completing before the end of the basic period.

The scheduler computes a latest possible starting time within the basic period for each performer by subtracting an estimate of the performer's running time from the duration of the basic period. Before invoking the performer, the conductor simply compares the current time to the latest start time stored in the schedule. If the current time is earlier than the latest start time, it invokes the performer.

The scheduler uses Chebyshev's inequality to compute a probabilistic estimate of a performer's running time from the performer's mean and standard deviation. Chebyshev's inequality does not produce the tightest possible estimates. However, it applies to all possible performers regardless of their true running time distribution, which is important because scheduling untrusted independently-authored performers requires a universal estimator.

The conductor suspends performers from the schedule if they fail to execute to completion. Consequently it uses the instantaneous admission control mechanism to protect a performer scheduled later in the basic period from a suspension forced on it by an earlier long-running performer. Performance tests show that LiquiMedia's instantaneous admission control mechanism prevents typical performers from executing past the end of the basic period.

The effectiveness of this mechanism greatly helps LiquiMedia to satisfy the principle of modularity by protecting performers from one another. Given its effectiveness, the instantaneous admission control mechanism may also be useful in other applications. It is simple, applicable to all possible performers and completely eliminates overtimes.

LiquiMedia's lifetime admission mechanism complements the instantaneous mechanism with admission control facilities in the non-realtime scheduler. It insures that all invocation sequences possible in a schedule exceed a specified firmness and corrects the schedule by removing performers if any do not. Its ability to perform admission control tests on arbitrary performers is central to LiquiMedia satisfying the principle of modularity.

The scheduler uses Chebyshev's inequality to estimate a schedule's running time at a given firmness from the entire schedule's mean and standard deviation for the same reasons as the

instantaneous mechanism. Chebyshev's inequality is conservative and universally applicable, so that it can estimate the running time of any potential combination of performers. It is also both simple to implement and efficiently computed.

On a set of 1662 schedules consisting of randomly selected typical performers, the scheduler, within 100 basic periods, successfully excluded all schedules whose firmness was less than the specified target of 0.9. This demonstrates that the LiquiMedia statistical scheduling mechanism can always detect typical schedules that cannot be reliably executed.

The scheduler uses feedback to correct schedules containing performers that run past the end of the basic period. Schedule feedback is guaranteed to reduce a schedule which fails its admission control test to one which will pass.

The scheduler removes performers from the schedule in order of increasing importance. LiquiMedia's implementation of importance is simple and easily implemented. Despite this, it insures that performers generating established media streams always have preference over newly scheduled performers whose output has not yet been segregated. This behaviour both reflects the results of perceptual psychology and helps LiquiMedia satisfy the principles of modularity and partitioning.

Hierarchical partitioning divides LiquiMedia into the conductor realtime child operating system and an arbitrary non-realtime child operating system. This design aspect makes it possible to retrofit the LiquiMedia architecture to an existing operating system. Building the LiquiMedia prototype on top of Solaris demonstrated this. As demonstrated by the LiquiMedia realtime JVM, hierarchical partitioning also permitted adding realtime facilities to an uncooperative environment such as the Java virtual machine.

LiquiMedia's conduit IPC mechanism connects the partitions. Conduits are an efficient asymmetric asynchronous IPC mechanism that permit exchanging atomic types such as integers between realtime performers and non-realtime tasks. Implementing a signal processing application comprised of both realtime and non-realtime media processing tasks for the LiquiMedia realtime JVM prototype demonstrated that conduits are an ideal design aspect for communication between operating system partitions.

8.3 Summary

In conclusion, this thesis has presented the LiquiMedia operating system architecture for multimedia. LiquiMedia has many features: principle-centered design, conduit IPC, ultra-fine grain performer-style tasks, hierarchical partitioning, distributed scheduling and statistical admission control. But most importantly, LiquiMedia takes advantage of the characteristics of a human audience to deliver a unique capability: it executes arbitrary untrusted independently-authored tasks at a quantifiable firmness.

Bibliography

[AAS97] T. F. Abdelzaher, Ella M. Atkins, and K. G. Shin. QoS negotiation in real-time systems and its application to automated flight control. In *Proceedings of the Real-Time Technology and Application Symposium (RTAS-97)*, pages 228–238, June 1997.

[AASD01] Ella M. Atkins, T. F. Abdelzaher, K. G. Shin, and E. H. Durfee. Planning and resource allocation for hard real-time, fault-tolerant plan execution. *Journal of Autonomous Agents and Multi-Agent Systems*, Mar-Apr 2001.

[AB98a] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. of the 19th IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1998.

[AB98b] Alia Atlas and Azer Bestavros. Statistical rate monotonic scheduling. Technical Report 1998-010, Boston University, 1998.

[AB99] Luca Abeni and Giorgio Buttazzo. QoS guarantee using probablistic deadlines. In *11th EuroMicro Conference on Real-Time Systems*, pages 242–249. IEEE Computer Society Press, 1999.

[ACD91a] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for probabilistic real-time systems. In *Automata, Languages and Programming*, pages 115–126, 1991.

[ACD91b] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Verifying automata specifications of probabilistic real-time systems. In *REX Workshop*, pages 28–44, 1991.

[ADS96] Ella M. Atkins, E. H. Durfee, and K. G. Shin. Plan development using local probabilistic models. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 49–56, August 1996.

[Adv03] Advanced Micro Devices Inc. *AMD Opteron Processor Data Sheet*, April 2003. Overview of AMD Opteron.

[AEW01] Philipp Ackermann, Dominik Eichelberg, and Bernhard Wagner. MET++. http://-www.ifi.unizh.ch/groups/mml/projects/met++/met++.html, August 2001.

[AL92] Martin Abadi and Leslie Lamport. An old-fashioned recipe for real time. Technical Report 91, Digital SRC, Palo Alto, California, 1992.

[ALW99] David R. Alexander, Douglas A. Lawrence, and Lonnie R. Welch. Feedback control resource management using *A Posteriori* workload characterizations. http://-citeseer.nj.nec.com/421506.html, 1999.

[APLW02] Luca Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *IEEE Real-Time Systems Symposium (RTSS)*, 2002.

[App88] Apple Computer, Addison-Wesley. *Inside Macintosh*, 1988.

[App95] Apple Computer Inc. QuickTime lawsuit information. http://support.info.apple.-com/aboutapple/lawsuit.html, November 1995.

[App97] Apple Computer Inc. *Rhapsody Operating System Software*, 1997.

[App00a] Apple Computer Inc. MacOS X aqua interface. http://www.apple.com/macosx/-aqua.html, January 2000.

[App00b] Apple Computer Inc. QuickTime 4 developer documentation. http://developer.-apple.com/techpubs/quicktime/, February 2000.

[App04] Apple Computer Inc. Mac OS X Tiger: Core Image. http://www.apple.com/mac-osx/tiger/core.html, January 2004.

[Axe] Jakob Axelsson. A hardware/software codesign approach to system-level design of real-time applications. http://citeseer.nj.nec.com/219788.html.

[Bac86] Maurice J. Back. *The Design of the Unix Operating System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Bar76] M. R. Barbacci. The symbolic manipulation of computer descriptions: ISPL compiler and simulator. Technical report, Department of Computer Science, CMU, Pittsburgh, 1976.

[Bar77] M. R. Barbacci. The ISPS computer description language. Technical report, Department of Computer Science, CMU, Pittsburgh, 1977.

[Bar93] A. Bartoli. Wide-address spaces: exploring the design space. *Operating systems Review*, 27(1):11–7, 1993.

[Bar97] Michael Barabanov. A Linux-based real-time operating system. Master's thesis, New Mexico Institute of Mining and Technology, 1997.

[BB02] Scott A. Banachowski and Scott A. Brandt. The BEST scheduler for integrated processing of best-effort and soft real-time processes. http://citeseer.ist.psu.edu/-banachowski02best.html, 2002.

[BBD+98] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *Linux Kernel Internals*. Addison Wesley Longman, Reading Mass., 1998.

[BN70] C. G. Bell and Andrew Newell. The PMS and ISP descriptive systems for computer structures. In *Spring Joint Computer Conference*, 1970.

[BN71] C. G. Bell and Andrew Newell. *Computer Structures: Readings and Examples*. McGraw-Hill Book Campany, New York City, 1971.

[Boe04] Hans Boehm. A garbage collector for C and C++. http://www.hpl.hp.com/-personal/Hans_Boehm/gc/, 2004.

[Bor90] Nathaniel S. Borenstein. *Multimedia applications development with the Andrew Toolkit*. Prentice Hall, Englewood Cliffs, N.J., 1990.

[BPM99] A. Bavier, L. Peterson, and D. Mosberger. BERT: A scheduler for best effort and realtime tasks. Technical Report TR-60299, Princeton University, March 1999.

[BR81] P. Bertelson and Radeau. Cross-modal bias and perceptual fusion with auditory-visual spatial discordance. *Perception and Psychophysics*, 29:578, 1981.

[Bre90] Albert S. Bregman. *Auditory Scene Analysis: The Perceptual Organization of Sound*. MIT Press, Cambridge, 1990.

[BS73] M. R. Barbacci and D. P. Siewiorek. Automated exploration of the design space for register transfer (rt) systems. In *First Annual Symposium on Computer Architecture*, 1973.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: methods, tools, and applications*. Addison-Wesley, Boston, Massachusetts, 2000.

[CGL83] Tony F. Chan, Gene H. Golub, and Randall J. LeVeque. Algorithms for computing the sample variance: Analysis and recommendations. *The American Statistician*, 37:242–247, 1983.

[Cha04] Ben Charny. Sprint PCS to talk wireless data. http://news.com.com/2100-1039-5137122.html, 2004.

[CLR90] Thomas H. Cormen, Chrarles E. Leiserson, and Ronald L. Rivest. *Introduction To Algorithms*. Cambridge, MIT Press, 1990.

[Cor70] Tom N. Cornsweet. *Visual Perception*. Academic Press, New York City, 1970.

[CS98] Jeffrey M. Capone and Ioannis Stavrakakis. Determining the call admission region for real-time heterogeneous applications in wireless TDMA networks. http://citeseer.nj.nec.com/73602.html, 1998.

[CS01] Marco Caccamo and Lui Sha. Aperiodic servers with resource constraints. In *Proceedings of the IEEE Real-Time Systems Symposium, London UK*, December 2001.

[CT00] David Cortesi and Susan Thomas. *REACT Real-Time Programmer's Guide*. Silicon Graphics, Inc., 2000. The IRIX realtime system.

[Dia00] Diamond Multimedia Inc. Diamond Rio portable music player. http://www.riohome.com/, February 2000.

[DN99] Edward N. Dekker and Joseph M. Newcomer. *Developing Windows NT Device Drivers*. Addison-Wesley, Reading, Massachusetts, 1999. How to write drivers for NT — filled with juicy tidbits.

[dNR00] Dionisio de Niz and Ragunathan Rajkumar. Chocolate: A reservation-based Real-Time Java environment on windows/NT. In *IEEE Real Time Technology and Applications Symposium*, pages 266–, 2000.

[Dul01] Margaret Dulat. The force buffer: A new architecture for force feedback. Master's thesis, University of Waterloo, 2001.

[Ert01] Anton Ertl. Threaded code. http://www.complang.tuwien.ac.at/forth/threaded-code.html, December 2001.

[FBL+03] Eric D. Feigelson, G. Jogesh Babu, Thomas Loredo, Fionn Murtagh, and Edward J. Wegman. StatCodes time series analysis. http://www.astro.psu.edu/statcodes/sc_timeser.html, July 2003.

[FM01] Alexandre R. J. Francois and Gerard G. Medioni. A modular software architecture for real-time video processing. In *ICVS*, pages 35–49, 2001.

[GC94] Berny Goodheart and James Cox. *The Magic Garden Explained*. Prentice Hall, Englewood Cliffs, N.J., 1994.

[GGV96] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, 1996. This system exists in the binary partitioning design space.

[GH01] Etienne M. Gagnon and Laurie J. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Java Virtual Machine Research and Technology Symposium*. USENIX, April 2001.

[GMSW89] W.M. Gentleman, S.A. MacKay, D.A. Stewart, and M. Wein. Using the Harmony operating system: Release 3.0. Technical Report ERA-377, NRCC No. 30081, National Research Council Canada, 1989.

[GRA89] James Gosling, David S.H. Rosenthal, and Michelle Arden. *The NeWS Book, An Introduction to the Network/extensible Window System*. SUN Technical Reference Library. Springer-Verlag, New York City, 1989.

[GSPW98] Ashvin Goel, David Steere, Calton Pu, and Jonathan Walpole. SWiFT: A feedback control and dynamic reconfiguration toolkit. Technical Report CSE-98-009, Oregon Graduate Institute, 1998. A general toolkit for using feedback control in operating systems.

[Han89] S. Handel. *Listening: an introduction to the perception of auditory events*. MIT Press, Cambridge, 1989.

[Har88] Harry G. Armstrong Aerospace Medical Research Laboratory, Wright-Patterson A.F.B., Ohio. *Engineering data compendium: human perception and performance*, 1988.

[HBJS91] James Hanko, David Berry, Thomas Jacobs, and Daniel Steinberg. Integrated multimedia at Sun Microsystems. In R. G. Herrtwich, editor, *Network and Operating System Support for digital Audio and Video*, Lecture Notes in Computer Science, Berlin, 1991.

[Hod94] Peter Hoddie. Somewhere in QuickTime: Basic movie playback support. *Develop, The Apple Technial Journal*, 18:22, March 1994.

[HS61] Ira J. Hirsh and Carl E. Sherrick. Perceived order in different sense modalities. *Journal of Experimental Psychology*, 62:423–432, 1961.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[Inc00a] Eonic Systems Inc. Virtuoso v.4.1 product data sheet. http://www.eonic.com/, March 2000.

[Inc00b] Precise Software Technologies Inc. Precise/MQX. http://www.psti.com/, March 2000.

[Int99] Intel Inc. *Intel Architecture Software Developers manual*, 1999.

[Jav00] JavaSoft Inc. Java media framework overview. http://java.sun.com/products/java-media/jmf/index.html, January 2000.

[JB95] Kevin Jeffay and David Bennett. A rate-based execution abstraction for multimedia computing. In *Network and Operating System Support for Digital Audio and Video*, pages 64–75, 1995.

[JG01] Kevin Jeffay and Steve Goddard. Rate-based resource allocation models for embedded systems. In *Embedded Software. First International Workshop, EMSOFT 2001*, number 2211 in Lecture Notes in Computer Science, Berlin, 2001. Springer-Verlag. Pages 1-7 of this article provide an excellent overview of the various forms of RBE and related forms of proportional share schedling.

[JIF+96] Michael B. Jones, Joseph S. Barrera III, Alessandro Forin, Paul J. Leach, and Marcel-Cătălin Roşu Daniela Roşu. An overview of the rialto real-time architecture. In *Seventh ACM SIGOPS European Workshop (SIGOPS 96)*, September 1996.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, New York City, 1996.

[JLDI95] Michael B. Jones, Paul J. Leach, Richard P. Draves, and Joseph S. Barrera III. Modular real-time resource management in the rialto operating system. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V),*, May 1995.

[JLS99] Jan Jonsson, Henrik Lonn, and Kang G. Shin. Non-preemptive scheduling of real-time threads on multi-level-context architectures. In *IPPS/SPDP Workshops*, pages 363–374, 1999.

[JMG+02] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002. USENIX, USENIX Association.

[Jon93] Michael B. Jones. Adaptive real-time resource management supporting composition of independently authored time-critical services. Technical Report TR-93-02, Microsoft Research, Microsoft Corporation, 1993.

[JSS91] Kevin Jeffay, Donald L. Stone, and F. Donelson Smith. Kernel support for live digital audio and video. In R. G. Herrtwich, editor, *Network and Operating System Support for digital Audio and Video*, Lecture Notes in Computer Science, Berlin, November 1991. Springer-Verlag.

[KC96] Robert Kroeger and William Cowan. LiquiMedia — a dynamically extensible cyclic executive. In Kevin Jeffay, editor, *IEEE RTSS: Workshop on Resource Allocation Problems in Multimedia Systems*, Washington, DC, December 1996. IEEE, IEEE Computer Society.

[Kop90] H. Kopetz. Event-triggered versus time-triggered real-time systems. In A. Karshmer J. Nehmer, editor, *Operating Systems of the 90s and Beyond*, number 563 in Lecture Notes in Computer Science, Berlin, July 1990. Springer-Verlag.

[Kro93] Robert J. Kroeger. Sonification: Adding streams of sound to a user interface. Master's thesis, University of Waterloo, 1993.

[Kro99] Robert Kroeger. Business directions — the second iteration. Technical report, LiquiMedia Inc., 1999.

[KSSR96] Hiroyuki Kaneko, John A. Stankovic, Subhabrata Sen, and Krithi Ramamritham. Integrated scheduling of multimedia and hard real-time tasks. In *IEEE Real-Time Systems Symposium 96*, 1996.

[LADS00] H. Li, Ella M. Atkins, E. Durfee, and K. G. Shin. Resource allocation for a limited real-time agent using a temporal probabilistic world model. In *Working Notes of the 2000 AAAI Spring Symposium on Real-Time Autonomous Systems*, 2000.

[Lee01] Elliot Lee. GNOME multimedia framework. http://developer.gnome.org/doc/-whitepapers/GMF/, August 2001.

[LHS+96] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *IEEE Real-Time Systems Symposium 96*, 1996.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LM95] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–461, 1995.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and implementation of the 4.3BSD UNIX operating system.* Addison-Wesley, Reading, Massachusetts, 1989.

[LMW96] Yu-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium 96*, 1996.

[Loc92] C. D. Locke. Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives. *Real-Time Systems*, pages 37–53, 1992.

[LSD89] John P. Lehoczky, Lui Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[LT96] Christopher J. Lindblad and David L. Tennenhouse. The VuSystem: A programming system for compute-intensive multimedia. *IEEE Journal of Selected Areas in Communications*, 14(7):1298–1313, 1996.

[Lyn02] LynuxWorks. Lynx OS: Real-time operating system. http://www.lynx.com-/products/lynxos/lynxos.php3, January 2002.

[Man67] G. K. Manacher. Production and stabilization of real-time task schedulers. *Journal of the ACM*, 14(3):439–465, July 1967.

[Men02] Mentor Graphics. VRTX: Real-time operating system. http://www.mentor-graphics.com/embedded/vrtxos/, January 2002.

[MEP01] Sorin Manolache, Petru Eles, and Zebo Peng. Memory and time-efficient schedulability analysis of task sets with stochastic execution times. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*. IEEE Computer Society Press, June 2001.

[MGH+95] James G. Mitchell, Jonathan J. Gibbons, Graham Hamilton, Peter B. Kessler, Yousef A. Khalidi, Panos Kougiouris, Peter W. Madany, Michael N. Nelson, Michael L. Powell, and Sanjay R. Radia. An overview of the Spring system. Technical report, Sun Microsystems Inc., 1995.

[MGSW96] K.G. Munhall, P. Gribble, L. Sacco, and M. Ward. Temporal constraints on the McGurk effect. *Perception and Psychophysics*, 58:351–362, 1996.

[Mic00a] Microsoft Inc. About DirectX. http://www.microsoft.com/directx/overview/ab-outdx.asp, January 2000.

[Mic00b] Microsoft Inc. COM component architecture. http://www.microsoft.com/com/, January 2000.

[Mic00c] Microsoft Inc. Microsoft Vizact 2000. http://www.microsoft.com/vizact/home.htm, January 2000.

[Mic00d] Realtime Microsystems. About SPARK. http://www.realtimemicrosystems.com/-page3.html, March 2000.

[Mic00e] Microware Inc. OS-9: Real-time operating system. http://www.microware.com-/Products/Rsrcs/WhitePapers/OS9_wp.pdf, August 2000.

[Mic04] Microsoft Inc., Redmond, WA. *Windows SDK*, 2004.

[Mok83] A. K. Mok. FUNDAMENTAL DESIGN PROBLEMS OF DISTRIBUTED SYSTEMS FOR THE HARD-REAL-TIME ENVIRONMENT. Technical Report MIT/LCS/TR-297, MIT, 1983.

[Mot98] Motorola Inc. *Semiconductor Product Brief DSP56651*, 1998.

[Mot00] Motorola Inc. Motorola Streammaster product. http://www.mot-sps.com/adc/-streamaster/, January 2000.

[MR01] Akihiko Miyoshi and Ragunathan Rajkumar. Protecting resources with resource control lists. In *Proceedings of IEEE RTAS'2001.*, 2001.

[MS85] M. McGrath and Q. Summerfield. Intermodal timing relations and audio-visual speech recognition by normal hearing adults. *Journal of the Acoustical Society of America*, 77(2):678–685, 1985.

[MST93] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves for multimedia operating systems. Technical Report CMU-CS-93-157, Carnegie Mellon University School of Computer Science, May 1993.

[MST94] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, May 1994.

[Nar99] Steve Naroff. Apple's unofficial assessment of JMF. Personal Communication, March 1999. Mr Naroff was management lead for Java and QuickTime development at Apple.

[NEC03] NEC. CiteSeer. http://citeseer.nj.nec.com/cs, July 2003.

[Nie99] Jason Nieh. *The design, implementation and evaluation of SMART: a scheduler for multimedia applications.* PhD thesis, Stanford University, 1999.

[NK91] J. Duane Northcutt and Eugene M. Kuerner. System support for time-critical applications. In R. G. Herrtwich, editor, *Network and Operating System Support for digital Audio and Video*, Lecture Notes in Computer Science, Berlin, 1991. Springer-Verlag.

[NL97] Jason Nieh and Monica S. Lam. Implementation and evaluation of SMART: A scheduler for multimedia applications. In *Sixteenth ACM symposium on Operating systems principles (SOSP'97)*, pages 184–197, St.Malo, France, October 1997.

[OR86] Ann O'Leary and Gillian Rhodes. Cross-modal effects on visual and auditory object perception. *Perception and Psychophysics*, 35:565–569, 1986.

[OR98] Shuichi Oikawa and Ragunathan Rajkumar. Linux/RK: A portable resource kernel in linux. In *19th IEEE Real-Time Systems Symposium*, December 1998.

[OS94] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-time Signal Processing*. Prentice Hall, Englewood Cliffs, N.J., 1994.

[PJG+01] Allalaghatta Pawan, Rakesh Jha, Lee Graba, Saul Cooper, Ionut Cardei, Mihaela Cardei, Vipin Ghopal, Sanjay Parthasanathy, and Saad Bedros. Real-time adaptive resource management. *IEEE Computer*, pages 99–100, July 2001.

[Poy96] Charles Poynton. Technical introduction to timecode. http://www.poynton.com/-notes/video/Timecode/index.html, 1996.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symposium*. USENIX, April 2001.

[QHG99] Gang Quan, Xiaobo (Sharon) Hu, and Garrison Greenwood. Hierarchical hardware/software partitioning, 1999.

[QNX00] QNX Inc. Qnx realtime os. http://www.qnx.com/products/os/qnxrtos.html, March 2000.

[Qua01] Qualcomm Inc. Qualcomm binary environment for wireless. http://www.qualcomm.com/brew, August 2001.

[Raj91] Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Norwell, MA, 1991.

[Raj00] Raj Rajkumar. Real-time and multimedia laboratory home page. http://www.cs.-cmu.edu/afs/cs.cmu.edu/project/art-6/www/rtmach.html, March 2000.

[red00] redhat.com. eCos — embedded cygnus operating system. http://www.cygnus.-com/ecos/wp.pdf, February 2000.

[Ree04] William J. Reed. The Pareto, Zipf and other power laws. http://linkage.rockefeller.edu/wli/zipf/reed01_el.pdf, August 2004.

[RJO+89] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, et al. Mach: A system software kernel. In *Proceedings of COMPCON '89*, Berlin, February 1989. Springer-Verlag.

[San03] Ingo Sander. *System Modeling and Design Refinement in ForSyDe*. PhD thesis, Royal Institute of Technology, Stockholm, 2003.

[SAWJ+96] Ion Stoica, H. Abdel-Wahab, Kevin Jeffay, S.K. Baruah, J.E. Gehrke, and C.G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, December 1996.

[SCG+00] Vijay Sundaram, Abhishek Chandra, Pawan Goyal, Prashant J. Shenoy, Jasleen Sahni, and Harrick M. Vin. Application performance in the QLinux multimedia operating system. In *ACM Multimedia*, pages 127–136, 2000.

[SFK97] Deszö Sima, Terence Fountain, and Péter Kacsuk. *Advanced Computer Architectures: A Design Space Approach*. Addison-Wesley, Harlow, England, 1997.

[SGG+99] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Operating Systems Design and Implementation*, pages 145–158. USENIX, 1999.

[Shn84] Ben Shneiderman. Response time and display rate in human performance with computers. *ACM Comput. Surv.*, 16(3):265–285, 1984.

[Sil99] Silicon Graphics, Inc. vwsnd.c. http://www.kernel.org, Linux 2.4, July 1999.

[SKC00] Youngsoo Shin, Daehong Kim, and Kiyoung Choi. Schedulability-driven performance analysis of multiple mode embedded real-time systems. In *Design Automation Conference*, pages 495–500, 2000.

[Sla97] Gerrit Slavenburg. *TriMedia TM1000 Data Book*. Philips Electronics North America Corporation, preliminary edition, 1997.

[SLS99] John A. Stankovic, Chenyang Lu, and Sang H. Son. The case for feedback control real-time scheduling. In *11th EuroMicro Conference on Real-Time Systems*, pages 11–20. IEEE Computer Society Press, 1999.

[Sol98] David A. Solomon. *Inside Windows NT*. Microsoft Press, Redmond, Washington, second edition edition, 1998.

[SPA93] SPARC International Inc. *The SPARC ARchitecture Manual*. Menlo Parc, CA, 1993.

[SPE95] SPEC Consortium. The benchmark page. http://performance.netlib.org/performance/html/spec.html, 1995.

[Spo03] John G. Spooner. PlayStation 3 chip nears completion. http://news.zdnet.co.uk/story/0,,t269-s2120395,00.html, July 2003.

[SR91] J. Stankovic and K. Ramamritham. The Spring kernel: A new paradigm for hard real-time operating systems. *IEEE Software*, 8(3):62–72, 1991.

[SSS+02] Deepak R. Sahoo, Swaminathan Sivasubramanian, M. V. Salapaka, G. Manimaran, and A. K. Somani. Feedback control for real-time scheduling. In *Proceedings of American Control Conference (ACC)*, 2002.

[Sun95a] Sun Microelectronics, http://www.sun.com/microelectronics/vis/download/vsdk/-dload.html. *Visual Instruction Set Users Guide*, 1995.

[Sun95b] SunSoft, Mountain View, California. *Multithreaded Programming Guide*, Solaris 2.5 edition, 1995.

[Sun95c] SunSoft, Mountain View, California. *priocntl manual Page*, Solaris 2.5 edition, January 1995.

[Sun00] Sun Microsystems Inc. JavaBeans component architecture. http://java.sun.com/-beans/, January 2000.

[Tay01] Wim Taymans. GStreamer. http://gstreamer.net/developers.shtml, August 2001.

[TDS+95] T. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L. Wu, and J. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Proceedings of the 1st IEEE Real-Time Technology and Applications Symposium*, pages 164–173. IEEE Computer Society Press, 1995.

[Tek95] A. Murat Tekalp. *Digital Video Processing*. Prentice Hall, Upper Saddle River, NJ, 1995.

[Ter01] Teralogic Inc. Teralogic TL880. http://www.teralogic-inc.com/Solutions/ICS/-tl880.asp, August 2001.

[Tex92] Texas Instruments. *Mwave Multimedia System Technical Brief*, 2647303-9721 edition, September 1992.

[Tex00] Texas Instruments Inc. TMS320 third-party program overview. http://www.ti.-com/sc/docs/general/dsp/third/index.htm, February 2000.

[TH97] T. Tan and W. Hsu. Scheduling multimedia applications under overload and non-deterministic conditions. In *Real-Time Technology and Application Symposium*, pages 178–183. IEEE Computer Society Press, 1997.

[VRT96] Lev Vaitzblit, Kadangode K. Ramakrishnan, and Percy Tzelnic. Scheduling and admission control policy for a continuous media server. United States Patent Number 55285513, June 1996.

[Wan95] John Wang. Somewhere in QuickTime: Choosing the right codec. *Develop, The Apple Technial Journal*, 21, March 1995.

[Wel99] Mark Wells. Architecture of Nokia 6000 series cellular phones. Personal Communication, October 1999. The prospective CEO of Zucotto systems. Had previously served as product manager for several Nokia cellular phones.

[Wic84] Christopher D. Wickens. *Engineering psychology and human performance.* C.E. Merrill, Toronto, 1984.

[Wil96] Paul R. Wilson. Uniprocessor garbage collection techniques. *ACM Computing Surveys (to appear)*, 1996.

[Win00] WindRiver. VxWorks 5.4 data sheet. http://www.wrs.com/products/html/-vxwks54_ds.html, February 2000.

[Wir77] Nicklaus Wirth. Toward a discipline of real-time programming. *Communications of the ACM*, 20(8):577–583, August 1977.

[Wir92] Niklaus Wirth. *Project Oberon: the design of an operating system and compilers.* ACM Press, Addison-Wesley, Reading Mass., 1992.

[WM78] Ronald E. Walpole and Raymond H. Myers. *Probability and Statistics for Engineers and Scientists.* Macmillan, New York City, 1978.

[Wol94] W. Wolf. Hardware-software co-design of embedded system. *Proceedings of the IEEE*, 82(7), July 1994.

[www00a] www.namesys.com. ReiserFS. http://www.namesys.com/, October 2000.

[www00b] www.rtlinux.org. RTLinux. http://www.rtlinux.org/rtlinux/index.html, February 2000.

[Yau99] David K. Y. Yau. ARC-h: Uniform CPU scheduling for heterogeneous services. In *ICMCS, Vol. 2*, pages 127–132, 1999.

[Yod96] Victor Yodaiken. Resource control in RT-Linux. In Kevin Jeffay, editor, *IEEE RTSS: Workshop on Resource Allocation Problems in Multimedia Systems*, Washington, DC, December 1996. IEEE, IEEE Computer Society Press.

[ZHS99] T. Zhou, X. Hu, and E. H.-M. Sha. A probabilistic performance metric for real-time system design. In *Proc. Int'l Workshop on Hardware/Software CoDesign*, pages 90–94, 1999.

Appendix A

Standard Nomenclature

Throughout this thesis I use the following mathematical representations for various objects. These are collected here for ease of reference when reading the document.

i The monotonically increasing positive integer index of each *release* of a task. In a global time-triggered system such as LiquiMedia, this is the index of a basic period. (c.f. 2.4, 2.2) The first basic period has the index 1.

p A task, most frequently found as an index on a per-task or per-performer property. We also make use of indicies x, y, z to refer to tasks as required.

$p(i)$ The result of executing task p in basic period i. (c.f. B.1)

$\mathcal{P}$ The set of schedulable realtime tasks in the system.

T_p A random variable expressing the running time of performer p.

$T_{\mathcal{P}}$ A random variable expressing the running time of all performers in some set of performers $\mathcal{P}$ (including a fixed overhead of $|\mathcal{P}|w_p + w_c$).

$\phi_p(t)$ The actual (and unknown) probability distribution of T_p:

$$P(T_p = t) = \phi_p(t)$$

$\Phi_p(t)$ The cumulative probability distribution of $\phi_p(t)$ where

$$\begin{aligned} P(T_p \leq t) &= \Phi_p(t) \\ &= \int_{-\infty}^{t} \phi_p(\tau) d\tau. \end{aligned}$$

$\phi_{\mathcal{P}}(t)$ The actual (and unknown) probability distribution of $T_{\surd}$:

$$P(T_{\mathcal{P}} = t) = \phi_{\mathcal{P}}(t)$$

$\Phi_{\mathcal{P}}(t)$ The cumulative probability distribution of $\phi_{\mathcal{P}}(t)$ where

$$\begin{aligned} P(T_{\mathcal{P}} \leq t) &= \Phi_{\mathcal{P}}(t) \\ &= \int_{-\infty}^{t} \phi_{\mathcal{P}}(\tau) d\tau. \end{aligned}$$

μ_p The mean of $\phi_p(t)$.

σ_p^2 The variance of $\phi_p(t)$.

(p, i) A tuple which uniquely identifies release i of task p.

$r_{p,i}$ The time of release i of task p. (c.f. 2.4, 2.2) In global time-triggered externalization, all tasks have identical release time r_i in each basic period.

$d_{p,i}$ The deadline for invocation i of task p. Successful realtime execution of task p requires that the entire execution of invocation i take place between $r_{p,i}$ and $d_{p,i}$. LiquiMedia's use of global time-triggered externalization gives the following simplification: $\forall p : d_{p,i} = d_i = r_{i+1} = r_i + T_B$.

$d'_{p,i}$ The relative deadline for invocation i of task p:

$$d'_{p,i} = d_{p,i} - r_{p,i}$$

If the operating system is rate monotonic, then the following condition holds: $\forall i : d'_p = d'_{p,i}$. In a globally time-triggered operating system, $\forall p : d'_p = T_B$. In an event-triggered operating system, the task specifies $d'_{p,i}$.

(p, i, j) A tuple uniquely identifying slice j of task p in release i. (c.f. 2.2)

$b_{p,i,j}$ The absolute start time of slice (p, i, j). When we are discussing performer-style task abstractions, we will drop the index j as it is an invariant 1.

$e_{p,i,j}$ The absolute end time of slice (p, i, j). When we are discussing performer-style task abstractions, we will drop the index j as it is an invariant 1.

$t_{p,i,j}$ The running time of slice j of task p in release i. Given that an operating system always executes a slice atomically, the following holds:

$$t_{p,i,j} = e_{p,i,j} - b_{p,i,j}$$

Also:

$$t'_{p,i} = \sum_{\forall j} t_{p,i,j}$$

If task p is a performer-style task, we drop the constant $j = 1$ subscript. Hence, $t_{p,i}$ is the actual time required to execute performer p to completion in release i. The actual observed running time of performer p in release i may be $t'_{p,i}$ where $t'_{p,i} < t_{p,i}$ if p misses its deadline and the operating systems has the incomplete soft realtime deadline sensitivity design aspect. (c.f. 2.3)

q A quantum: no slice runs longer than q. The operating system using the preemptive thread task abstraction insures that $q \geq \forall_{p,i,j} t_{p,i,j}$. (c.f. 2.2.2)

$\mathsf{wcet}(p)$ The worst case execution time (WCET) of performer p defined by:

$$\text{wcet}(p) = \max_{\forall i}(t_{p,i})$$

$\mathsf{stat}(p)$ LiquiMedia's statistical estimate of performer p's running time defined by:

$$P(T_p < \text{stat}(p)) = P_c$$

$t'_{p,i}$ The actual time that the operating system spent executing task p between $r_{p,i}$ and $d_{p,i}$ For a task to have successfully executed in realtime $t'_{p,i} = t_{p,i}$.

t_i The actual time needed to execute all the performers in $\text{path}(1, i)$. Note that

$$t_i > \sum_{p \in \text{path}(1,i)} t_p$$

because the conductor's per-performer overhead w_p and fixed overhead w_c are both greater than 0.

q A time quantum. Several scheduling functions use q to bound the running time of a slice: $\forall p, i, j : t_{p,i,j} \leq q$. (c.f. 2.5.4)

$\mathcal{S}$ The scheduling function

$$\mathcal{S} : \mathcal{P} \times \mathbf{N} \times \mathbf{N} \rightarrow \mathcal{P}$$

returns the next task to execute. The function $\mathcal{S}$'s domain is a slice tuple of performer index, basic period and slice index. We drop the constant slice index of 1 when discussing scheduling functions for performer style tasks. (c.f. 2.5)

$m_{p,i}$ The mean of performer p's running times at the end of basic period i.

$s^2_{p,i}$ The variance of performer p's running times at the end of basic period i.

$n(p,i)$ The total number of times that performer p has executed successfully at the end of basic period i.

$\lceil m_{p,i,j} \rceil^{\alpha}$
An upper bound (c.f. equation B.6) on μ_p such that:

$$P(\mu_p < \lceil m_{p,i,j} \rceil^{\alpha}) = \alpha$$

$m_{\mathcal{P},i}$ The mean of the conductor's running times in basic period i.

$\lceil m_{\mathcal{P},i,j} \rceil^{\alpha}$
An upper bound (c.f. equation B.6) on $\mu_{\mathcal{P}}$ such that:

$$P(\mu_{\mathcal{P}} < \lceil m_{\mathcal{P},i,j} \rceil^{\alpha}) = \alpha$$

$s^2_{\mathcal{P},i}$ The variance of the conductor's running times in basic period i.

$\mathsf{free}(\mathcal{G})$ The estimated fraction of the basic period left unallocated after executing the longest path in $\text{path}^*(\text{first}(\mathcal{G}); \mathcal{G})$.

$M_{p,i,j}$ The conductor-maintained sum of $t_{p,i}$ at the end of basic period i since the last reset of the statistical profile data in basic period j. (c.f. section B.3.6)

$S_{p,i,j}$ The conductor-maintained sum of $t^2_{p,i}$ at the end of basic period j since the last reset of the statistical profile data in basic period j. (c.f. section B.3.6)

$\mathsf{N(p,i,j)}$ The integer-valued number of times that the conductor has successfully executed performer p at the end of basic period i since the scheduler last reset the statistical summary information in basic period j. (c.f. section B.3.6)

t_c The current time.

T_B The duration of a single basic period in a global time-triggered RTOS: $T_B = r_{i+1} - r_i$.

T_L The onset latency of an information stream. (c.f. Chapter 4)

$i(p)$ The index of a basic period where the conductor first invokes performer p.

$i(\mathcal{G})$ The basic period in which the conductor first uses schedule $\mathcal{G}$.

$N(\mathcal{G}, i, j)$
The number of basic periods in which the conductor has used schedule $\mathcal{G}$ in basic periods $i, \ldots, j$. Note that

$$0 \leq N(\mathcal{G}, i, j) \leq \min(j - i(\mathcal{G}), H)$$

P_c The minimum probability of completion or the *firmness* of an operating system.

P_g The minimum probability that $\hat{\phi}_p(t)$ accurately infers $\phi_p(t)$.

$\mathsf{lt}(p, P_c)$
The latest relative start time for performer p such that it has a probability greater than P_c of completing before the end of the basic period:

$\mathcal{P}(t)$ The finite set of all live realtime tasks at time t. A live task is one which has been released but has not yet completed execution.

$$\mathcal{P}(t) = \{\forall p \in \mathcal{P} : \exists i : t \in [r_{p,i}, d_{p,i}) \text{ and } \exists j : e_{p,i,j} > t\}$$

$\mathcal{G}$ An arbitrary schedule graph. (c.f. B.3.3)

$\mathcal{G}_c(i)$ The schedule graph used by the conductor. (c.f. B.3.3)

$V(\mathcal{G})$ The vertices in a schedule graph $\mathcal{G}$. This is a always a set of performers. (c.f. B.3.3)

$E(\mathcal{G})$ The edges in the schedule graph where $E(\mathcal{G}) = \{(x, y) : x, y \in V(\mathcal{G})\}$. (c.f. B.3.3)

$\mathsf{first}(\mathcal{G})$ The first node in $\mathcal{G}$.

$\mathsf{Adj}(p; \mathcal{G})$
The adjacency set of some performer p in graph $\mathcal{G}$ is as follows:

$$\mathrm{Adj}(p; \mathcal{G}) = \mathrm{Adj}^1(p; \mathcal{G}) = \{x \in V(\mathcal{G}) : (p, x) \in E(\mathcal{G})\}$$

Note that the implementation views this as a *sequence* though, it is logically a set (c.f. B.3.3). Let $\mathrm{Adj}_i(p; \mathcal{G})$ be element i of $\mathrm{Adj}(p; \mathcal{G})$ when viewed as a sequence.

$\mathsf{Adj}_k(p; \mathcal{G})$
The k-th element of $\mathrm{Adj}(p; \mathcal{G})$. (c.f. B.3.3)

$\mathsf{Adj}(p)$ A notational convenience in the interests of brevity:

$$\mathrm{Adj}(p) = \mathrm{Adj}(p; \mathcal{G}_c)$$

$\mathsf{Adj}^n(p; \mathcal{G})$
The recursively applied Adj() operator. (c.f. B.3.3)

$\mathsf{Adj}^*(p; \mathcal{G})$
All possible performers which could be executed after p in an arbitrary traversal of the schedule graph:

$$\mathrm{Adj}^*(p; \mathcal{G}) = \mathrm{Adj}^n(p; \mathcal{G}) \text{for } n \text{ such that } \mathrm{Adj}^n(p; \mathcal{G}) = \mathrm{Adj}^{n+1}(p; \mathcal{G})$$

(c.f. B.3.3)

$\mathsf{path}(p, i)$

The sequence of performers beginning with p that the conductor attempts to execute in basic period i in the absence of errors where

$$\text{path}(p, i) \in \text{path}^*(p; \mathcal{G}_\rfloor)$$

. Note that $\text{path}(p, i)$ cannot be determined in advance as each p controls the result of $\mathcal{S}(p, i)$ for p where $|\text{Adj}(p; \mathcal{G}_c)| > 1$. (c.f. B.3.3)

The members of $\text{path}(p, i)$ are the same as $\mathcal{P}(b_{p,i})$. However, we use the path notation as it preserves the performer order of execution and better reflects the underlying structure of a schedule graph.

$\mathsf{path}^*(p; \mathcal{G})$

The set of all possible paths that rooted at p over schedule graph $\mathcal{G}$.

$\mathsf{path}_i(p; \mathcal{G})$

The i-th path in the set $\text{path}^*(p; \mathcal{G})$.

$\mathsf{prgs}(p)$ The number of first-tier overtimes that performer p has experienced since it last executed. value of performer p. (c.f. section B.3.5)

$\mathsf{defr}(p)$ The total number of first-tier overtimes that p has experienced since beginning execution. (c.f. section B.3.5)

H The highest schedule age. (c.f. section B.3.6)

ν_i The ith normal score. (c.f. E.2)

w_f The context switch time of a Solaris preemptive thread. (c.f. 6.4.2)

w_p The per-performer overhead of the conductor. (w is for wait or wastage.) (c.f. B.4.5)

w_c The conductor's fixed overhead independent of the number of performers executed. (c.f. B.4.5)

Appendix B

Architecture Details

Chapter 4 provides an overview of LiquiMedia's architecture. This appendix presents the architecture at a level of detail sufficient for implementors. The system interfaces are taken from the LiquiMedia realtime JVM because these interfaces are an improvement on the prototype's lower level C interfaces.

B.1 Performers

LiquiMedia has the performer task abstraction for realtime tasks. Performers provide a lightweight task primitive suitable for satisfying the principle of ultra-fine granularity realtime. A performer is an instance of a class which implements the `Performer` interface in listing B.1.

As discussed in 2.2, the variable p represents a performer. The set $\mathcal{P}$ contains all scheduled performers in the system (and is defined more precisely below in Section B.3.3). Executing some performer p corresponds to the conductor invoking the `perform(PerformerState ps)` method on an code object corresponding to p.

Let $p(i)$ be the value returned by performer p when executed in basic period i. $p(i)$ is either one of the pre-defined status codes `OK` or `REMOVE` or the successor of p in the schedule graph. Section B.3.4 formalizes the scheduling function's use of $p(i)$.

Each implementation of `Performer` must also implement the following methods:

`public double getEstimatedMean( PerformerState pstate);`

This method provides the LiquiMedia scheduler with its initial estimate of a performer's running time: $m_{p,i(p)-1}$.

Listing B.1: Performer Interface

```
package com.liquimedia.api;

public interface Performer {

    public int perform( PerformerState pstate);
    public double getEstimatedMean( PerformerState pstate);
    public double getEstimatedSdev( PerformerState pstate);
    public Conduit getNotifierConduit();

    public final static int OK = 0;
    public final static int REMOVE = -1;

}
```

`public double getEstimatedSdev( PerformerState pstate);`
: This method provides the scheduler with its initial estimate of the standard deviation of a performer's running time: $s_{p,i(p)-1}$.

`public Conduit getNotifierConduit();`
: This method either returns `null` or a performer-writable conduit. The conductor uses this conduit to notify other application components if the performer has experienced an error such as an overtime.

The conductor provides each performer with read-only state and timing information via the `PerformerState` interface shown in listing B.2. Section B.3.1 precisely defines the results of these functions.

B.2 Statistical Inference

LiquiMedia uses statistical inference to compute a bound on a performer's running time from a sample of previously observed running times. Scheduling performers using this bound permits safely scheduling untrustworthy independently authored performers as the bound depends only on the performers' actual behaviour and not on any unreliable performer-provided declaration.

A performer p has a running time represented by the random variable T_p. T_p is the elapsed real time needed for performer p to execute to completion. Knowing in advance all the possible values for the random variable T_p would permit perfect scheduling. Such foreknowledge is impossible. Instead, scheduling p in realtime requires an upper bound on T_p.

Listing B.2: PerformerState Interface

```
package com.liquimedia.api;

public interface PerformerState {
    double getDITmean();
    double getDITsdev2();
    int getAge();
    int getProgress();
    int getOvertimeCount();
    int getBasicPeriod();
    double now();
    double cycleStart();
    double start();
}
```

Bounds can be guaranteed or probabilistic. The worst case execution time (WCET) bound is guaranteed as wcet(p) satisfies equation B.1 — it exceeds all possible values of T_p. Hard realtime scheduling requires knowing a task's WCET bound. Conversely, a probabilistic bound is not guaranteed. The probabilistic bound stat(p) satisfies equation B.2 — it has only a probability P_c of exceeding all possible values of T_p.

$$\text{wcet}(p) \geq \max_{\forall i}(t_{p,i}) \tag{B.1}$$

$$P(T_p < \text{stat}(p)) = P_c \tag{B.2}$$

LiquiMedia can compute the probabilistic bound stat(p) from a sampling of T_p and so does not rely on a performer to correctly declare its running time. Instead, LiquiMedia schedules performers based only on their actual behaviour and so can safely schedule untrustworthy independently authored performers.

B.2.1 Chebyshev's Inequality

LiquiMedia uses Chebyshev's inequality [WM78] to compute the bound stat(p) because the inequality permits computing a bound without knowing T_p's associated probability distribution function (PDF).

Knowing T_p's PDF $\phi_p(t)$ permits solving

$$P(T_p \leq t) = \int_0^{\text{stat}(p)} \phi_p(t)dt. \tag{B.3}$$

for a value of stat(p) which is considerably smaller than the value obtained from Chebyshev's equation. However, this method of computing stat(p) is impractical because an operating system cannot automatically determine $\phi_p(t)$. The statistically-scheduled RTOSs described in [TDS+95, AB98b, AB99, AB98a, ZHS99] sidestep the impossible task of determining $\phi_p(t)$ by requiring p to provide its PDF to the system scheduler. However an untrustworthy performer p can even less be relied upon to provide a correct definition of its PDF than it can be to provide its wcet(p) so this "solution" precludes satisfying the principle of modularity.

Mechanically computing the true $\phi_p(t)$ is impossible and cannot safely be left to the developers of performer p. Chebyshev's inequality is the only satisfactory method for computing stat(p) unless a satisfactory approximation to $\phi_p(t)$ exists and can be used in its place. An approximating distribution $\hat{\phi}_p(t)$ must be both applicable to all possible performers and easily computable. Given the ease with which it can be computed, I explored the use of the normal distribution as the approximating distribution. Appendix E presents the results: the normal distribution did not adequately approximate $\phi_p(t)$ for any test performers.

Instead, LiquiMedia takes the only practical choice to support of independently-authored performers. It uses Chebyshev's inequality:

$$P(\mu_p - k\sigma_p < T_p < \mu_p + k\sigma_p) \geq 1 - \frac{1}{k^2} \tag{B.4}$$

to compute stat(p) because equation B.4 holds for *all* possible performers regardless of their true probability distribution functions. Substitute P_c and re-express Chebyshev's inequality as follows:

$$P(\mu_p - k_c\sigma_p < T_p < \mu_p + k_c\sigma_p) \geq P_c$$

where

$$k_c = \sqrt{\frac{1}{1 - P_c}}.$$

Then,

$$\text{stat}(p) = \mu_p + k_c\sigma_p. \tag{B.5}$$

B.2.2 Sample Statistics Corrections

Equation B.5 requires the mean μ_p and standard deviation σ_p of $\phi_p(t)$. With $\phi_p(t)$ unknown, these values are not available. LiquiMedia approximates them from the sample mean $m_{p,i}$ and the standard deviation $s_{p,i}$. These two sample statistics include contributions from samples $t_{p,i(p)}, \ldots, t_{p,i}$ where $i(p)$ is the first basic period in which the conductor executes performer p and $t_{p,i}$ is the actual running time of performer p when executed in basic period i.[1]

[1] Only performers from every path through the schedule graph will have been executed in every basic period $i(p), \ldots, i$ (c.f. B.4).

Estimation theory provides the following probabilistic bound on the value of μ_p:

$$P(\mu_p < \lceil m_{p,i} \rceil^{\alpha}) = \alpha \tag{B.6}$$

where

$$\lceil m_{p,i} \rceil^{\alpha} = m_{p,i} + z_{\alpha} \frac{\sigma_p}{\sqrt{i - i(p) + 1}} \tag{B.7}$$

and where z_α is the value of the standard normal distribution with an area of α on the left [WM78].

Appendix E shows that the population of performer running times is non-normal. Consequently, $\lceil m_{p,i} \rceil^{\alpha}$ is a poor estimate for a performer which has executed for less than 30 basic periods.[2] For larger numbers of executions, it provides a reasonable estimate. Fortunately, almost all schedule re-verifications include run time data from more than 30 invocations of a performer.[3] For this reason, it is also safe to assume that $\sigma_p \approx s_{p,i}$ for all i. This assumption is highly desirable because it eliminates the need to code a complex correction and thereby simplifies the implementation of the scheduler.

Equation B.7 permits computing a more accurate value of stat(p) than equation B.5. Choose probabilities α and β such that $P_c = \alpha\beta$ and replace P_c in equation B.5 with β. Set

$$\alpha = \beta = \sqrt{P_c}$$

Then,

$$\text{stat}(p) = m_{p,i} + k_c s_{p,i} + z_{\alpha} \frac{s_{p,i}}{\sqrt{i - i(p) + 1}} \tag{B.8}$$

where

$$k_c = \sqrt{\frac{1}{1 - \alpha}}.$$

However, the small size of this correction permits simplifying the implementation by omitting it from the scheduler. A firmness of $P_c = 0.99$ is adequate for a multimedia application. This firmness gives $\alpha = 0.9949$ and $z_\alpha = 2.57$. Comparing the terms from equation B.5 after only one second of execution:

$$z_{\alpha} \frac{s_{p,i}}{\sqrt{i - i(p) + 1}} = .29 s_{p,i}$$

and

$$k_c s_{p,i} = 14.12 s_{p,i}$$

shows that the correction makes only a 2% difference. Such a small difference does not justify the implementation burden. Consequently, the scheduler computes stat(p) with equation B.5.

[2] This approximation is valid even for a non-normal distribution when at least 30 samples points are available [WM78, page 195].

[3] Performers exist to generate information streams for human observers. Useful streams have lifespans measured in minutes to hours — many thousands of performer invocations.

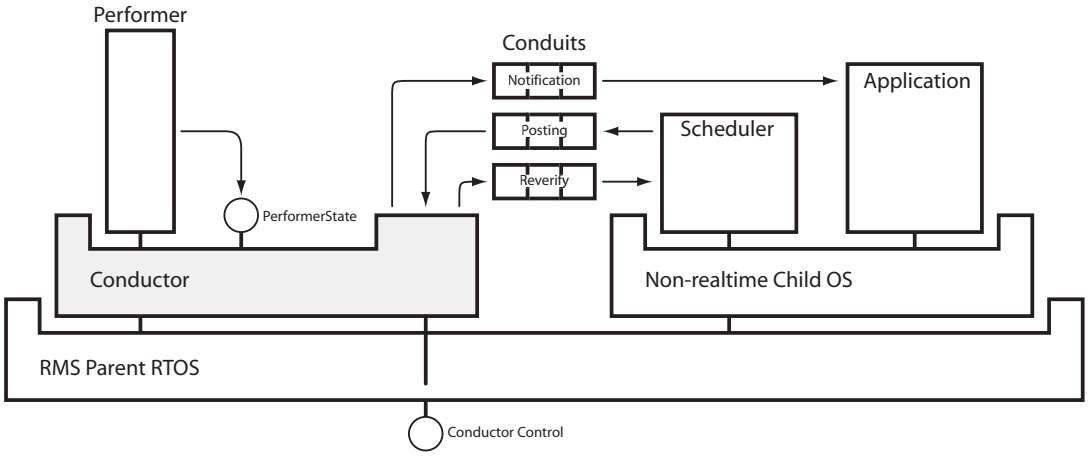

Figure B.1: The figure shows the conductor's relationship to the remainder of LiquiMedia.

The estimate stat(p) in equation B.5 can be efficiently computed. Further, it is independent of a performer's implementation. Consequently, using Chebyshev's inequality for statistical admission control enables LiquiMedia to provide admission test independently authored realtime fragments.

B.3 The Conductor

The conductor constitutes one of LiquiMedia's two sub-operating systems. It is a realtime RMS-scheduled task. It executes in response to a hardware timer with a period of T_B. The conductor dispatches correct performers sequentially as specified by the scheduling graph. The prototype permits only a single conductor. Chapter 7 discusses the challenges of supporting multiple conductors executing in parallel.

B.3.1 Interfaces

Figure B.1 shows the conductor's four major sub-interfaces: schedule management, performer state inquiries, performer notifications and control. The sub-interfaces are categorized by their function, need for asynchrony and the accessor's required privilege.

The schedule management interface consists of the *reverify conduit* and the *post conduit*. These two conduits provide a private two-way asynchronous connection between the conductor and the scheduler.

The conductor requests a re-verification of the current schedule $\mathcal{G}_c(i)$ by writing its pointer to the reverify conduit. After verifying a schedule's correctness, the scheduler writes it to

the post conduit. The conductor switches to the newest schedule in the post conduit at the beginning of the next appropriate basic period.

LiquiMedia keeps the actual schedule graph and per-performer summary statistics in memory shared between the scheduler and the conductor. The access patterns discussed in Section B.4.6 permit the conductor and scheduler to asynchronously share schedule graph data.

The `PerformerState` inquiry interface provides each performer with read-only access to the conductor's accumulated per-performer scheduling information. The conductor passes an implementation of this interface when it calls a performer's `perform` method. The interface is synchronous and private.

Listing B.2 shows the `PerformerState` interface. The purpose of its member functions are defined below for performer p when executed in basic period i.

`double getDITmean();`
: Returns $m_{p,i-1}$ in nanoseconds. $m_{p,i-1}$ is computed on demand using equation B.15 described in Section B.3.6.

`double getDITsdev();`
: Returns $s_{p,i-1}$ in nanoseconds. $s_{p,i-1}$ is also computed on demand using equation B.16 described in Section B.3.6.

`int getAge();`
: Returns $I(p, i-1)$ which is the number of times that the conductor has successfully executed performer p in basic periods $i(p), \ldots, i-1$.

`int getProgress();`
: Returns the performer's progress value prgs(p) which is 1+ the number of times that the conductor has failed to execute p since it last successfully invoked p. In normal operation, prgs(p) = 1.

`int getDeferralCount();`
: Returns defr(p) which is the total number of deferrals experienced by this performer since $i(p)$.

`double now();`
: Returns the current time t_c in nanoseconds.[4]

`double cycleStart();`
: Returns the start of the current basic period r_i.

`double start();`
: Returns the time $b_{p,i}$ when the conductor invoked p.

[4]All timing routines have an infelicity imposed by the time needed to actually execute the measurement functions. See Section 5.2.5.

A performer can use the methods of `PerformerState` to adjust its behaviour appropriately. Consider the following examples.

- A video decompressor detects that it has been deferred in the previous basic period when `getProgress()` returns 2. Fortunately it maintains a buffer of two frames so does not drop any frames. It decompresses the current frame and selects an alternate branch of the schedule graph with its return code where a second performer refills the buffer with the result of decompressing an additional video frame.[5]

- A realtime video compositing package contains a performer which renders particle system sparks and blends them onto a video stream. The number of particles varies randomly. The performer constrains its running time by rendering particles until

$$\texttt{now()} - \texttt{start()} > .2T_B.$$

The conductor uses the performer notification interface to notify applications of performer suspensions. Each performer instance must implement `getNotifierConduit()`. If this method returns a non-null conduit, the conductor writes a message containing an error code and a reference to the offending performer when suspending the performer for any of the following reasons: the performer returned an invalid value, it caused an overtime, its prgs(p) exceeded a configurable maximum or, its defr(p) exceeded a configurable maximum.

These notification messages permit applications to adjust their schedule graphs. Consider the following examples.

- An audio effects application consists of some number of performers performing audio filtering and some supervisory composers. One composer listens for messages from the audio-processing performers. If it receives any, it displays a warning dialog to the user. The dialog indicates that the application is having difficulty meeting its processor needs and permits the user to alter the relative importance of applications in the system.

- In a realtime 3D simulation such as a game, the renderer is divided into a pair of performers: the "core" renderer and the "glitz" renderer. The core renderer runs first and the glitz renderer runs immediately afterward. The core renderer listens for overtime error messages from the glitz renderer. Figure B.2 shows the application's schedule graph with three levels of "glitz" rendering. The application wishes to maximize the glitz level.

  If the core renderer receives a notification indicating that the glitz renderer has experienced an overtime in the previous basic period, its return code specifies the next lower cost glitz performer and requests that a non-realtime application thread re-submits the suspended high glitz performer to the schedule.

[5] While a performer object may only occur once in any path through the scheduling graph, two performer objects may execute the same underlying code.

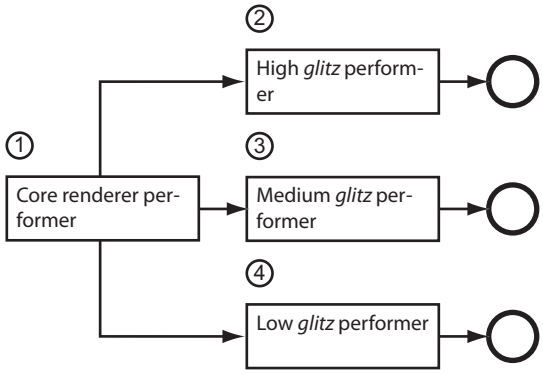

Figure B.2: Glitz Level Schedule Paths

Writing notification messages contributes to the conductor's per-performer overhead. The principle of ultra-fine granularity realtime requires minimizing this overhead. Consequently, the conductor writes at most one fixed-size message per performer invocation.

Finally, the conductor control interface permits starting and suspending the operation of the conductor by toggling the value of the debugging flag. This interface is private and synchronous. It is available to debug the conductor. With the conductor suspended, it is possible to inspect its internal state.

B.3.2 Conductor Operation Overview

Algorithm 1 provides an overview of the conductor's operation. The conductor loops forever. Unless the conductor has been suspended through the control interface, it receives a basic period interrupt every T_B seconds. Upon receiving the basic period interrupt, the conductor first determines if it was executing when the basic period interrupt fired and then handles any overtime that may have occurred.

The conductor then traverses the schedule graph from beginning to end and invokes each performer on the traversed path. For each performer, the loop body contains the following phases: instantaneous admission control, performer execution, data collection and selecting the next performer in the graph. The following sections discuss the conductor's operation in greater detail.

B.3.3 Schedule Graphs

LiquiMedia uses distributed scheduling. Schedules are directed acyclic graphs (DAGs) where each vertex contains a performer. A performer specifies its successor to the conductor with its

Algorithm 1 Conductor Operation

basic period $i \leftarrow 0$
debugging-halt $\leftarrow 0$
loop
  Wait for a basic period interrupt
  Reentrant interrupt handler, phase one (c.f. section B.3.7)
  $i \leftarrow i + 1$
  $r_i \leftarrow t_c$
  if debugging-halt $= 1$ **then**
    Block basic period interrupt
    Break to debugger
  end if
  if basic period $i - 1$ completed normally **then**
    $\mathcal{G}_c \leftarrow$ newest posted schedule.
  end if
  $p \leftarrow \text{first}(\mathcal{G}_c(i))$ (c.f. section B.3.3)
  while $p \neq \emptyset$ **do**
    $b_{p,i} \leftarrow t_c$
    if p passes instantaneous admission control test (c.f. B.3.5) **then**
      invoke performer p
      $e_{p,i} \leftarrow t_c$
      $t_{p,i} \leftarrow e_{p,i} - b_{p,i}$
      $r \leftarrow p$'s return value
      Update statistical profile data (c.f. section B.3.6)
    end if
    $p \leftarrow \mathcal{S}(p, i)$ (c.f. section B.3.4)
  end while
  $t_i \leftarrow t_c - r_i$
  Update whole-schedule profile data (c.f. section B.3.6)
  if verify $\mathcal{G}_c$ at $N(\mathcal{G}_c, i(\mathcal{G}_c), i)$ **then**
    request verification of $\mathcal{G}_c$
  end if
end loop

return code. The sequence of performers executed in a single basic period is a *path* through the schedule graph.

Precisely specifying the conductor's operation requires a formal definition of a schedule graph $\mathcal{G}$. $\mathcal{G}$ has vertices $V(\mathcal{G})$ and edges $E(\mathcal{G})$. Each vertex is a performer. first($\mathcal{G}$) is the first performer in $\mathcal{G}$. Define $\text{Adj}(p;\mathcal{G})$ to be the adjacency set of some performer $p \in V(\mathcal{G})$ with the following properties.

(i) $\text{Adj}(p;\mathcal{G}) = \{x \in V(\mathcal{G}) : (p,x) \in E(\mathcal{G})\}$

(ii) The set $\text{Adj}(p;\mathcal{G})$ contains $n = |\text{Adj}(p;\mathcal{G})|$ performers which can each be uniquely specified via a subscript:[6]

$$\text{Adj}(p;\mathcal{G}) = \{\text{Adj}_1(p;\mathcal{G}), \text{Adj}_2(p;\mathcal{G}), \ldots, \text{Adj}_n(p;\mathcal{G})\}.$$

(iii) Generalize $\text{Adj}(p;\mathcal{G})$ to operate over a subset $\mathcal{R}$ of $\mathcal{G}$ as follows:

$$\text{Adj}(\mathcal{R};\mathcal{G}) = \bigcup_{p\in\mathcal{R}} \text{Adj}(p;\mathcal{G}).$$

(iv) Given the above generalization, Adj can be recursively applied. Let $\text{Adj}^i(p;\mathcal{G})$ be the i-th application such that

$$\text{Adj}^n(p;\mathcal{G}) = \begin{cases} \text{Adj}(\text{Adj}^{i-1}(p;\mathcal{G});\mathcal{G}) & \text{if } i > 1 \\ \text{Adj}(p;\mathcal{G}) & \text{if } i = 1. \end{cases}$$

(v) $\text{Adj}^*(p;\mathcal{G}) = \text{Adj}^n(p;\mathcal{G})$ for an n such that $\text{Adj}^n(p;\mathcal{G}) = \text{Adj}^{n+1}(p;\mathcal{G})$. That is, $\text{Adj}^*(p;\mathcal{G})$ is all possible performers which could be executed after p in all possible traversals of the schedule graph $\mathcal{G}$.

Define $\mathcal{G}_c(i)$ to be the conductor's current schedule graph in basic period i. Then,

$$\mathcal{P} = V(\mathcal{G}_c(i))$$

for any basic period i. In the interests of brevity, also define:

$$\text{Adj}(p) = \text{Adj}(p;\mathcal{G}_c(i))$$

and

$$\text{Adj}^*(p) = \text{Adj}^*(p;\mathcal{G}_c(i))$$

over all basic periods.

The scheduling graph $\mathcal{G}_c(i)$ must satisfy the following properties.

[6]The C implementation indicies start at 0.

(i) There are no loops: $\forall p \in \mathcal{P} : p \notin \text{Adj}^*(p)$.

(ii) The scheduling graph has an unambiguous unique entry point $\text{first}(\mathcal{G}_c(i))$ and is fully connected:
$$\mathcal{P} = \{\text{first}(\mathcal{G}_c(i))\} \cup \text{Adj}^*(\text{first}(\mathcal{G}_c(i))).$$

(iii) There is always an end: $\exists p \in \mathcal{P} : \text{Adj}^*(p) = \emptyset$.

As discussed in Section B.4, the scheduler verifies that every schedule graph that it provides to the conductor has these properties.

B.3.4 Scheduling Function

Because LiquiMedia has distributed scheduling (c.f. 2.5), applications control the result of the conductor's evaluation of the scheduling function $\mathcal{S}$ in two fashions. First, as discussed below in Section B.4, non-realtime application code collectively defines $\mathcal{G}_c(i)$ and thereby determines the contents of the set $\text{Adj}(p)$. Second, because $\text{Adj}(p)$ can contain more than one performer, p specifies the next performer in $\text{Adj}(p)$ that the conductor should execute next. If p specifies an incorrect performer, the conductor executes the first element in $\text{Adj}(p)$.

Formally, LiquiMedia's scheduling function is:

$$\mathcal{S}(p,i) \leftarrow \begin{cases} \emptyset & \text{if } |\text{Adj}(p, \mathcal{G}_c(i))| = 0 \\ \text{Adj}_1(p, \mathcal{G}_c(i)) & \text{if } p(i) \notin [1, |\text{Adj}(p, \mathcal{G}_c(i))|] \\ \text{Adj}_{p(i)}(p, \mathcal{G}_c(i)) & \text{if } p(i) \in [1, |\text{Adj}(p, \mathcal{G}_c(i))|] \end{cases} . \tag{B.9}$$

The conductor invokes the scheduling function repeatedly in each basic period. Let the following recurrence represent these repeated applications:

$$\begin{aligned} \mathcal{S}^1(p,i) &= \mathcal{S}(p,i) \\ \mathcal{S}^n(p,i) &= \mathcal{S}(\mathcal{S}^{n-1}(p,i), i). \end{aligned}$$

Then, define $\text{path}(p,i)$ to be the sequence of performers beginning with p that the conductor attempts to execute in basic period i in the absence of errors:

$$\text{path}(p,i) = p, \mathcal{S}(p,i), \mathcal{S}^2(p,i), \ldots, \mathcal{S}^n(p,i)$$

where $\mathcal{S}^{n+1}(p,i) = \emptyset$.

Because $\mathcal{S}(p,i)$ depends on $p(i)$, $\text{path}(p,i)$ cannot be determined without executing all of its constituent performers. Consequently, let $\text{path}^*(p;\mathcal{G})$ be the set of all possible paths through the schedule $\mathcal{G}$ starting at performer p. Obviously, $\text{path}(p,i) \in \text{path}^*(p;\mathcal{G}_c(i))$. Unlike $\text{path}(p,i)$, the scheduler can determine $\text{path}^*(p;\mathcal{G})$ from a traversal of $\mathcal{G}$.

B.3.5 Instantaneous Admission Control

To reduce the probability that a performer will experience an overtime, the conductor has an instantaneous admission control mechanism. Listing 2 defines the mechanism precisely. Before invoking some performer p in basic period i, the conductor verifies that

$$t_c - r_i \leq \mathrm{lt}(p, P_c) \tag{B.10}$$

where

$$\mathrm{lt}(p, P_c) = T_B - \mathrm{stat}(p).$$

The scheduler pre-computes the latest relative start time $\mathrm{lt}(p, P_c)$ each time that it creates a new schedule graph.

The conductor maintains the number of successive deferrals $\mathrm{prgs}(p)$ and the total number of deferrals $\mathrm{defr}(p)$. It increments these counters every time that p fails to satisfy equation B.10 or when $p \in \mathrm{Adj}^*(x, i)$ and performer x has generated an overtime in a basic period i. Should either of $\mathrm{prgs}(p)$ and $\mathrm{defr}(p)$ exceed a configurable upper bound, the conductor suspends p and writes a message to p's notification conduit.

Algorithm 2 Instantaneous Admission Control Mechanism

Require: current performer p, its latest start time $\mathrm{lt}(p, P_c)$,
  if $t_c - r_i \leq \mathrm{lt}(p, P_c)$ **then**
    return passed.
  else
    $\mathrm{prgs}(p) \leftarrow \mathrm{prgs}(p) + 1$
    if $\mathrm{prgs}(p) >$ maximum consecutive first-tier overtimes **then**
      suspend p
      send error to notification conduit (c.f. section B.3.1)
      request verification of $\mathcal{G}_c(i)$
    end if
    $\mathrm{defr}(p) \leftarrow \mathrm{defr}(p) + 1$
    if $\mathrm{defr}(p) >$ maximum life-time first-tier overtimes **then**
      suspend p
      send error to notification conduit (c.f. section B.3.1)
      request verification of $\mathcal{G}_c(i)$
    end if
    return failed.
  end if

B.3.6 Statistical Profile Data

Successful statistical scheduling requires that both the conductor and LiquiMedia's scheduler have access to accurate summary statistics. The scheduler must compute these statistics from accumulated profile data. Naive use of floating point arithmetic in these computations produces incorrect results.

In [CGL83], Chan, Golub and LeVeque provide a comprehensive description of the numerical problems and present several possible solutions. LiquiMedia uses a combination of fully accurate integer arithmetic and Chan, Golub and LeVeque's pair-wise algorithm to compute each performer's the mean and standard deviation.

The conductor collects the number of successful executions $N_{p,i,j}$, the sum $M_{p,i,j}$, and sum of squared running times $S_{p,i,j}$ for each successful execution of a performer. It collects this data in machine integers. As discussed further in section B.4, the scheduler resets these values at least every H basic periods to prevent the integers from overflowing. These values are computed as shown in equation B.11 for a schedule graph $\mathcal{G}$ reset in basic period j.

$$
\begin{aligned}
M_{p,i,j} &= \begin{cases} 0 & \text{if } i < j \\ M_{p,i-1,j} + t_{p,i} & \text{if } i \geq j,\ p \in \text{path}(\text{first}(\mathcal{G}), i) \\ M_{p,i-1,j} & \text{if } i \geq j,\ p \notin \text{path}(\text{first}(\mathcal{G}), i) \end{cases} \\
S_{p,i,j} &= \begin{cases} 0 & \text{if } i < j \\ S_{p,i-1,j} + t_{p,i}^2 & \text{if } i \geq j,\ p \in \text{path}(\text{first}(\mathcal{G}), i) \\ S_{p,i-1,j} & \text{if } i \geq j,\ p \notin \text{path}(\text{first}(\mathcal{G}), i) \end{cases} \\
N_{p,i,j} &= \begin{cases} 0 & \text{if } i < j \\ N_{p,i-1,j} + 1 & \text{if } i \geq j,\ p \in \text{path}(\text{first}(\mathcal{G}), i) \\ N_{p,i-1,j} & \text{if } i \geq j,\ p \notin \text{path}(\text{first}(\mathcal{G}), i) \end{cases}
\end{aligned}
\tag{B.11}
$$

The conductor uses only integer arithmetic in the collection of $M_{p,i,j}$, $S_{p,i,j}$ and $N_{p,i,j}$ for two reasons. First, the use of integer arithmetic insures exact computation. Second, not using floating point arithmetic operations in the conductor facilitates extending an existing kernel with LiquiMedia-style realtime support. For example, the Solaris, Linux and Windows NT kernels optimize system-call performance by not saving floating point state across a system-call [DN99, BBD+98, GC94].

Much of the thesis treats $t_{p,i}$ as a real number quantity of time. However, this treatment is incompatible with its use in pure-integer computation and is not strictly correct. Each $t_{p,i}$ is the difference between a hardware clock-cycle counter before and after executing p. Consequently, $t_{p,i}$ is a unit-less number of machine cycles and is indeed usable in pure-integer computation. LiquiMedia resolves this quibble by requiring the existence of a proportionality constant a which converts the count $t_{p,i}$ into a quantity of time. The constant a has units of seconds per cycle and is typically the clock cycle duration of the host hardware.

Value	Bound	Prototype
$M_{p,i,j}$	$\lceil \log_2 \frac{HT_B}{a} \rceil$	37
$S_{p,i,j}$	$\lceil \log_2 H \left(\frac{T_B}{a}\right)^2 \rceil$	61
$I_{p,i,j}$	$\lceil \log_2 H \rceil$	14

Table B.1: Overflow-free Integer Sizes

Values of H (basic periods), T_B (time) and the proportionality constant a (time/cycle) determine machine integer sizes whose range guarantees that $M_{p,i,j}$, $S_{p,i,j}$ and $N_{p,i,j}$ do not overflow. Table B.1 contains formulas for the needed bit width and the actual widths needed in the `liqui` prototype where a is 1ns, T_B is 13ms and H is 9123.

A single 64-bit integer can accommodate any one of `liqui`'s summary statistics without overflow. With processor clock speeds exceeding 1GHz, this is no longer the case. Fortunately, integers larger that the host's native machine integer size have only a small performance penalty. For example, on the Intel X86 architecture, a single addition operation takes $2n$ assembly language instructions to add together two $32n$-bit integers [Int99]. A high-quality assembly language implementation on a 3GHz Pentium4 could compute the 96-bit $S_{p,i,j}$ in 6 (short) clock cycles.

The scheduler uses Chan, Golub and LeVeque's pair-wise approach to combine the $M_{p,i,i(\mathcal{G})}$, $S_{p,i,i(\mathcal{G})}$ and $N_{p,i,i(\mathcal{G})}$ collected for schedule $\mathcal{G}$ with floating point summary statistics $n_{p,i(\mathcal{G})-1}$, $m_{p,i(\mathcal{G})-1}$ and $s^2_{p,i(\mathcal{G})-1}$. Let the float() operator convert an integer value to a floating point number. The scheduler computes the total number of times that some performer p has executed successfully:

$$n(p,i) = n(p,i(\mathcal{G})-1) + \text{float}(N_{p,i,i(\mathcal{G})}) \tag{B.12}$$

and letting

$$x = n(p,i(\mathcal{G})-1) \tag{B.13}$$

$$y = \text{float}(N_{p,i,i(\mathcal{G})}) \tag{B.14}$$

in the interests of compactness, it computes the following recurrences for the mean and variance:

$$m_{p,i} = \frac{1}{x+y}\left(x m_{p,i(\mathcal{G})-1} + \text{float}(M_{p,i})\right) \tag{B.15}$$

$$\begin{aligned} s^2_{p,i} = \frac{1}{x+y}\Bigg(& x s^2_{p,i(\mathcal{G})-1} \\ & + \text{float}\left(S_{p,i} - \frac{1}{y}M^2_{p,i}\right) \\ & + \frac{x}{y(x+y)}\left(\frac{y}{m_{p,i(\mathcal{G})-1}} - \text{float}(M_{p,i})\right)^2\Bigg). \end{aligned} \tag{B.16}$$

This approach has an error bound of $\kappa u \log(x+y)$ for machine accuracy u where κ has the following approximation

$$\kappa \approx m_{p,i}/s_{p,i}.$$

when $s_{p,i}$ is small compared to $m_{p,i}$ and the number of basic periods is large. Because both of these conditions hold, the pair-wise algorithm is numerically sound and generates summary statistics sufficiently accurate for successful statistical scheduling.

As discussed in section B.1, each performer must provide estimates to the scheduler of its mean and standard deviation. Equations B.15 and B.16 show that LiquiMedia does not require that these initial estimates be correct. From the first schedule re-verification onward, LiquiMedia schedules p only on its *actual* running times. In order to support the execution of untrusted independently-authored realtime performers, LiquiMedia trusts application code as little as possible. Conscientious programmers should however still provide best-guess values of a performer's running time, typically through experience with running their code.

Internally, LiquiMedia may, as required in the interests of an efficient implementation, use a platform-specific unit for all timing statistics. Consequently, all application-level interfaces retrieve or set time values in a double precision floating point number of nanoseconds. I named this format Device Independent Time or DIT. Each implementation must provide appropriate conversion code from DIT to Device Dependent Time or DDT values. This conversion is trivial in the prototype as its DDT format is an integer count of nanoseconds.

B.3.7 Reentrant Operation

Sometimes the conductor has not completed executing when it receives the next basic period interrupt. Consequently, the conductor must support re-entrant operation. This complicates the conductor's implementation. If still executing, the conductor can be in one of four different states upon receipt of a basic period interrupt.

(i) Interrupt $i+1$ takes place before the execution of the first performer.

(ii) Interrupt $i+1$ takes place between the execution of p and $\mathcal{S}(p,i)$.

(iii) Interrupt $i+1$ takes place while executing performer p. In this case, p has experienced an overtime.

(iv) Interrupt $i+1$ takes place after the completion of all performers in $\text{path}(1,i)$.

Ideally, the conductor would block the basic period interrupt except while actually executing performers. This eliminates all but case (iii). Not all hardware and parent RTOSs permit this simplification. Further, disabling and re-enabling the basic period interrupt for every

performer invocation can impose performance penalties not compatible with the principle of ultra-fine granularity. For example, disabling the basic period interrupt in the LiquiMedia prototype takes time similar to the running time of an audio shaping performer. Consequently, algorithm 3 and algorithm 4 define the conductor's recovery from all four types of reentrant invocation.

Assume that performer p attempts to execute past r_{i+1} in basic period i. Then, the conductor should increment the deferral count for all performers $x \in \text{path}(\mathcal{S}(p,i),i)$ at the beginning of basic period $i+1$. However, $\text{path}(\mathcal{S}(p,i),i)$ cannot be determined. Instead, as shown in listing 4, the conductor notifies all $x \in \text{Adj}^*(p;\mathcal{G}_c(i))$ by incrementing $\text{prgs}(x)$ and $\text{defr}(x)$ and writing a message to x's notification conduit.

As a result, having experienced a reentrant interrupt in basic period i, the conductor notifies all performers

$$y \in \text{Adj}^*(p;\mathcal{G}_c(i)) - \text{path}(\mathcal{S}(p,i),i)$$

in error. LiquiMedia requires applications to correctly handle spurious notifications.

Two reasons reduce the needed development effort. First, the conductor never notifies *simple* performers in error where a simple performer is one which is in every path through the current schedule graph.

Second, every complex application scheduling graph such as the one shown in Figure 4.7 has a simple first performer (1 in the diagram). This performer must already specify which of performers $\text{Adj}(1) = \{2,3,8,10\}$ should execute next. The additional development effort needed for performer 1 to manage the wrongly incremented progress values of its successors is small.

Deferrals pose an additional problem. For example, if performer 1 in figure 4.7 experiences a deferral, which of performers should $2,3,8,10$ execute next? In the LiquiMedia prototype, the conductor executes $\text{Adj}_1(p)$ after deferring p. The conductor proceeds identically when it has suspended p in a previous basic period. An application can take advantage of this policy to define a schedule graph which handles performer suspensions or deferrals.

A related issue with overtimes is whether or not suspending one of an application's performers should result in suspending all of them. Suspending sibling performers can be desirable but is complex to implement and can significantly increase the conductor's execution overhead. For these reasons, the LiquiMedia prototype does not suspend the sibling performers of a performer that has experienced an overtime. The notification conduit provides adequate support for applications to submit updated schedule graphs. Because a complete schedule graph is regularly reassembled from per-application contributions as discussed below in Section B.4.4, this simpler approach does not leave orphaned performers in the schedule beyond the next reverification.

Algorithm 3 Reentrant Interrupt Handling

Require: basic period blocked from wait

in basic period i
nsf $\leftarrow$ the number of stack frames
$\mathcal{P} \leftarrow \emptyset$
if nsf $= 2$ **then** {A basic period interrupt occurred in the conductor}
  if $p = \emptyset$ **then** {after executing all performers case (iv)}
    $\mathcal{P} \leftarrow \emptyset$
  else if $p = \text{first}(\mathcal{G}_c(i))$ **then** {before executing any performers, case (i)}
    $\mathcal{P} \leftarrow V(\mathcal{G}_c(i))$
    if before incrementing i **then**
      $i = i + 1$
    end if
    reentrant interrupt notification of $\mathcal{P}$
  else if case (ii) and p executed **then**
    $\mathcal{P} \leftarrow \text{Adj}^*(p)$
    reentrant interrupt notification of $\mathcal{P}$
  else if case (ii) and p not executed **then**
    $\mathcal{P} \leftarrow \{p\} \cup \text{Adj}^*(p)$
    reentrant interrupt notification of $\mathcal{P}$
  end if
else if nsf > 2 **then** {p was running at r_{i+1}, case (iii).}
  suspend p
  if p has a notification conduit **then**
    write an overtime notification message to p's notification conduit
  end if
  $\mathcal{P} \leftarrow \text{Adj}^*(p)$
  reentrant interrupt notification of $\mathcal{P}$
end if
Enable basic period interrupt.
Pop nsf $- 1$ stack frames and restart loop.

Algorithm 4 Reentrant Interrupt Notification

```
    for x ∈ 𝒫 do
      prgs(x) ← prgs(x) + 1
      defr(x) ← defr(x) + 1
      if prgs(x) > maximum consecutive deferrals then
 5:     suspend x.
        if x has a notification conduit then
          write x and maximum consecutive deferral error to notification conduit (c.f. section B.3.1)
        end if
      else if defr(x) > maximum life-time deferrals then
10:     suspend x.
        if x has a notification conduit then
          write x and maximum life-time error to notification conduit (c.f. section B.3.1)
        end if
      else
15:     if x has a notification conduit then
          write x, "deferred by other's second-tier" message to x's conduit (c.f. section B.3.1)
        end if
      end if
      request re-verification of 𝒢_c(i)
20: end for
```

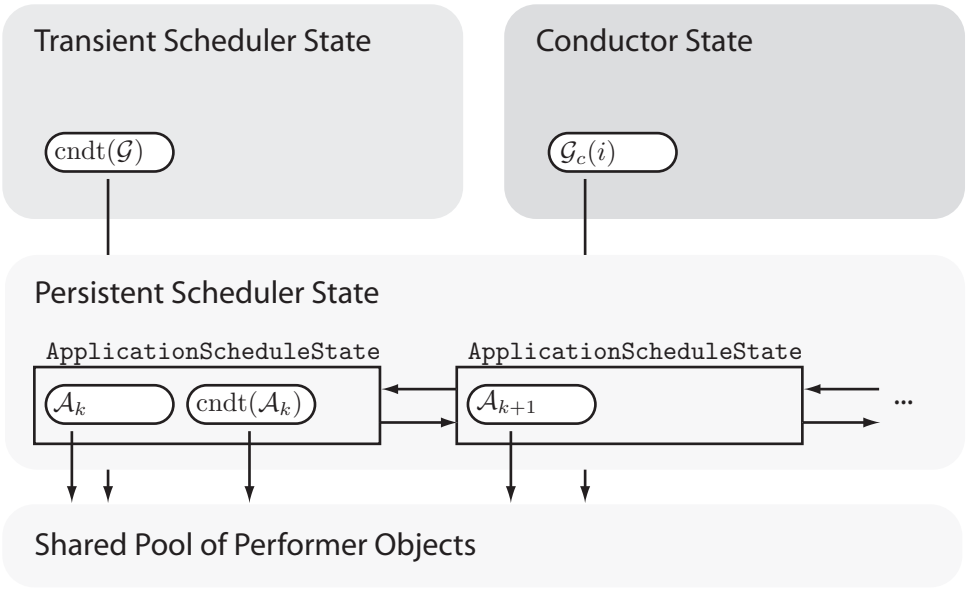

Figure B.3: The figure shows the data structures shared between the conductor, scheduler and applications.

B.4 Scheduler

LiquiMedia's scheduler prepares schedules from which the conductor executes performers. The scheduler is a continuously running composer task started when LiquiMedia boots. Its operation can be divided into four stages: waiting to service re-verification requests, assembling per-application schedule graphs into a single complete schedule graph, admission-testing this single schedule graph and, should the new schedule be admissible, delivering it to the conductor.

B.4.1 Schedule Data Structures

The scheduler's operation depends on the underlying data structures shown in Figure B.3: a shared pool of schedule graphs, performer objects and `ApplicationScheduleState` objects. These data structures can be categorized by their presence in the four stages of the scheduler's execution. The scheduler maintains its *persistent* data structures while waiting for re-verification requests. It augments this data with additional *transient* data structures while assembling and admission-testing a new schedule graph.

The scheduler's persistent data structures consists of a doubly-linked list of `ApplicationScheduleState` objects. There is one such object for each running application which has

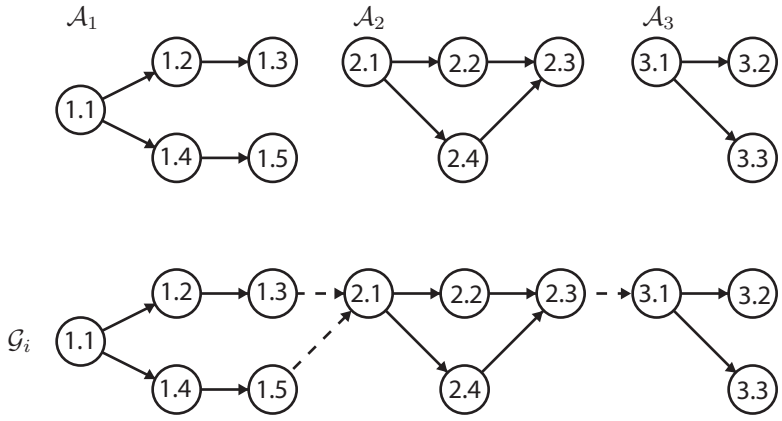

Figure B.4: An example of a well-formed schedule both as per-application fragments and connected by the scheduler into a graph $\mathcal{G}_i$.

invoked the `restructure` method discussed below in Section B.4.4. The `ApplicationScheduleState` object for application k contains the schedule graph $\mathcal{A}_k$. $\mathcal{A}_k$ is the k-th application's contribution to the most recently admitted schedule.

The scheduler's transient data structures depend on the origin of a re-verification request. If the scheduler awakens as a result of a request from application k, its transient data structures include two additional schedule graphs. First, $\arg(\mathcal{A}_k)$ is the schedule graph prepared by application k and passed by reference to the `restructure` method. Second, the `restructure` method prepares $\mathrm{cndt}(\mathcal{A}_k)$ by copying $\arg(\mathcal{A}_k)$ and then inserting it into the `ApplicationScheduleState` object for application k.

The `restructure` method copies $\arg(\mathcal{A}_k)$ into $\mathrm{cndt}(\mathcal{A}_k)$ to insure that an application cannot avoid admission testing by modifying a previously submitted shared schedule graph $\arg(\mathcal{A}_k)$. Section B.4.4 discusses the operation of the `restructure` method in greater detail.

In any re-verification, the scheduler's transient data consists of the schedule graph $\mathrm{cndt}(\mathcal{G})$. As described in Section B.4.3, the scheduler assembles the per-application schedule graphs into this graph and then applies the lifetime admission control test described in Section B.4.5. If scheduler can admit $\mathrm{cndt}(\mathcal{G})$, it posts the schedule to the conductor.

Figure B.4 shows an example of the assembly process for a re-structuring invoked by the conductor. The scheduler has constructed $\mathrm{cndt}(\mathcal{G})$ from the three application schedules $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$.

Each successful admission results in a new schedule in a sequence:

$$\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_j.$$

The conductor's current schedule $\mathcal{G}_c(i)$ is taken from these sequence of posted schedules. Section B.4.6 discusses the relationship between successive admitted schedules.

All of these different schedule graphs draw their nodes from an array of `s_Performer` objects shared between the conductor and scheduler. `s_Performer` objects have a sub-structure imposed by the scheduler and conductor's different access patterns to the `s_Performer`'s fields. The scheduler initializes all of the fields. Once the `s_Performer` object belongs to a posted schedule, fields are either entirely read-only, conductor read-write and scheduler read-only or are read-write for both the conductor and scheduler.

The first two cases require no special treatment. The third case jeopardizes correct operation of the conductor and scheduler unless there is a method for safe shared access. Only the statistical profile data defined in equation B.11 is read-write by both the scheduler and conductor. Fortunately, its stereotyped access patterns permit a simple solution to the shared access problem.

The statistical profile information is initialized by the scheduler, repeatedly updated by the conductor and eventually read and then reset by the scheduler. Multiple separate statistical profile objects solve the shared access problem. The scheduler first prepares a new statistical profile object. Then, it atomically exchanges this object's reference for the `s_Performer`'s existing reference. This atomic exchange operation gives the conductor a new zeroed statistical profile object to update while the scheduler computes aggregate statistical summary values from the preserved snapshots.

B.4.2 Graph Construction

In the `liqui` prototype's Java interface, applications schedule performers using the application programming interface (API) discussed below in Section B.4.4. The methods comprising this API take a reference to a schedule graph as an argument. An application k defines its schedule graph argument $\arg(\mathcal{A}_k)$ by creating instances of the `ScheduleGraph` class. This class has the following methods.

`ScheduleGraph(Performer p, String name)`

This constructor creates a new `ScheduleGraph` instance $\mathcal{A}$ with

$$V(\mathcal{A}) = \{p\}$$

and naming it `name`. The `name` parameter is optional. Each of the other methods below operate on an instance $\mathcal{A}$ of `ScheduleGraph`.

`ScheduleGraph set(ScheduleGraph` $\mathcal{A}_1$`, int i)`

Adds an edge to $E(\mathcal{A})$ such that

$$\mathrm{Adj}_i(\mathrm{first}(\mathcal{A}); \mathcal{A}) = \mathrm{first}(\mathcal{A}_1).$$

`ScheduleGraph get(int i)`
: Returns the `ScheduleGraph` wrapping performer $\mathrm{Adj}_i(\mathrm{first}(\mathcal{A}); \mathcal{A})$.

`ScheduleGraph add(ScheduleGraph` $\mathcal{A}_1$`)`
: Adds an edge $(\mathrm{first}(\mathcal{A}, \mathrm{first}(\mathcal{A}_1))$ to $E(\mathcal{A})$ at the lowest unused index. This method is a convenience for the common case of a single outbound edge or the addition of adjacencies in order.

`Performer getPfmer()`
: Returns the performer $\mathrm{first}(\mathcal{A})$.

`ScheduleGraph find(String name)`
: Returns the `ScheduleGraph` with the given name.

An application developer uses these methods to define an application schedule graph. For example, the execution of the code in Listing B.3 creates the application schedule graph $\mathcal{A}_2$ shown in Figure B.4.

B.4.3 Schedule Assembly

All schedule re-verification operations require the scheduler to assemble $\mathrm{cndt}(\mathcal{G})$ from per-application contributions. This section precisely defines the assembly process. Schedule assembly assumes that each application schedule graph satisfies the properties discussed in section B.3.3.

Let the $\oplus$ operator concatenate two schedule graphs such that if

$$\mathcal{G} = \mathcal{A}_j \oplus \mathcal{A}_k$$

then

$$\begin{aligned} V(\mathcal{G}) &= V(\mathcal{A}_j) \cup V(\mathcal{A}_k) \\ E(\mathcal{G}) &= E(\mathcal{A}_j) \cup E(\mathcal{A}_k)) \\ &\cup \bigcup_{\forall p \in \mathrm{Leaf}(\mathcal{A}_j)} \{(p, \mathrm{first}(\mathcal{A}_k))\} \end{aligned}$$

where $\mathrm{Leaf}(\mathcal{A}_j)$ is a function returning the leaf nodes of $\mathcal{A}_j$ and is defined as follows:

$$\mathrm{Leaf}(\mathcal{G}) = \{\forall x \in V(\mathcal{G}) : \forall y \in V(\mathcal{G}), (x, y) \notin E(\mathcal{G})\}.$$

Given the above definition of $\oplus$, a candidate schedule $\mathcal{G}$ assembled from k application schedule graphs is

$$\mathcal{G} = \bigoplus_{\kappa=1}^{k} \mathcal{A}_\kappa.$$

Listing B.3: Use of `ScheduleGraph` Class

```
import com.liquimedia.api;

class MakeGameSchedule {
  class CoreRenderer implements Performer {
    /* 2.1 */
  }
  class HighGlitzRenderer implements Performer {
    /* 2.2 */
  }
  class LowGlitzRenderer implements Performer {
    /* 2.4 */
  class GlobalLightingPass implements Performer {
    /* 2.3 */
  }

  ScheduleGraph sg;
  public MakeGameSchedule() {
    ScheduleGraph glp = createScheduleGraph(
      new GlobalLightingPass());

    sg = createScheduleGraph(new CoreRenderer()).add(
      createScheduleGraph(new HighGlitzRenderer()).add(
        glp)).add(
      createScheduleGraph(new LowGlitzRenderer()).add(glp));

    // Attempt to schedule sg...
  }
}
```

Listing B.4: Scheduler Interface

```
package com.liquimedia.api;

public interface Scheduler {
    public final static int FAILED = 0;
    public final static int POSTED = 1;
    public final static int TOOSLOW = 2;
    public final static int BAD_GRAPH  = 3;

    public int restructure(ScheduleGraph k, Conduit c);
    public int restructure(ScheduleGraph k );
}
```

The assembly algorithm itself consists of fundamental graph algorithms: Leaf($\mathcal{G}$) is the set of leaves of a spanning tree produced by a breadth first search of $\mathcal{G}$ while actually connecting the application graphs consists of adding the necessary edges to the candidate graph's adjacency lists.

B.4.4 Restructuring

An application modifies the scheduling of its performers with one of two different interfaces. The `Scheduler` interface shown in Listing B.4 permits an application to restructure its schedule graph while the `PrivilegedScheduler` interface discussed below permits a privileged application to modify the relative importance of applications.

Graph Restructuring

Application code adds, alters or removes performers from the schedule with the `restructure( ScheduleGraph k, Conduit c)` method from the `Scheduler` interface. Typically, an application takes the following steps to modify its schedule of running performers.

(i) The application creates or modifies an existing application candidate schedule graph arg($\mathcal{A}_k$) out of instances of `ScheduleGraph` class as discussed in section B.4.1.

(ii) The application invokes the `restructure` method with argument arg($\mathcal{A}_k$) and an optional conduit on which the application can receive messages from the scheduler.

(iii) The `restructure` mechanism executes Algorithm 5.

(iv) The application handles any errors which might have occurred as indicated by `restructure`'s return code.

(v) The application waits on conduit `c` or any notification conduits that it may have specified for its performers.

Algorithm 5 `restructure` Operation

Require: Argument $\arg(\mathcal{A}_k)$, optional conduit c
  if $\arg(\mathcal{A}_k)$ fails to satisfy graph properties discussed in Section B.3.3 **then**
    return `BAD_GRAPH`
  end if
  if application did not provide conduit c **then**
    $c \leftarrow$ temporary conduit from scheduler to application
  end if
  for $p \in V(\arg(\mathcal{A}_k)) - V(\mathcal{A}_k)$ **do**
    allocate `s_Performer` object p.
    initialize `s_Performer` p.
  end for
  $V(\text{cndt}(\mathcal{A}_k)) \leftarrow \emptyset$
  $E(\text{cndt}(\mathcal{A}_k)) \leftarrow \emptyset$
  for $p \in V(\arg(\mathcal{A}_k))$ **do**
    $V(\text{cndt}(\mathcal{A}_k)) \leftarrow V(\text{cndt}(\mathcal{A}_k)) \cup$ `s_Performer` corresponding to p
  end for
  $E(\text{cndt}(\mathcal{A}_k)) \leftarrow E(\arg(\mathcal{A}_k))$
  post $\text{cndt}(\mathcal{A}_k)$ to scheduler
  response $\leftarrow c$ (blocks on c)
  return response

The `restructure` method executes synchronously inside the application thread. Upon invocation, it first verifies that $\arg(\mathcal{A}_k)$ is a valid application candidate schedule graph. If so, `restructure` creates an internal copy $\text{cndt}(\mathcal{A}_k)$. LiquiMedia confines these operations to the submitting application's thread. This insures that a hostile application cannot interfere with the scheduler by submitting a deliberately malformed application candidate schedule. Finally, the `restructure` method makes a restructuring request by passing $\text{cndt}(\mathcal{A}_k)$ to the scheduler and waits for the scheduler's response.

The scheduler responds to the restructuring request by re-assembling all of the application schedules other than application k's candidate. It assembles them in order of descending importance. Lastly, it appends the candidate schedule $\text{cndt}(\mathcal{A}_k)$ after all of the other un-modified application schedule graphs. The scheduler then applies LiquiMedia's lifetime admission control test on the resulting candidate schedule graph. If the candidate is admissible, the scheduler

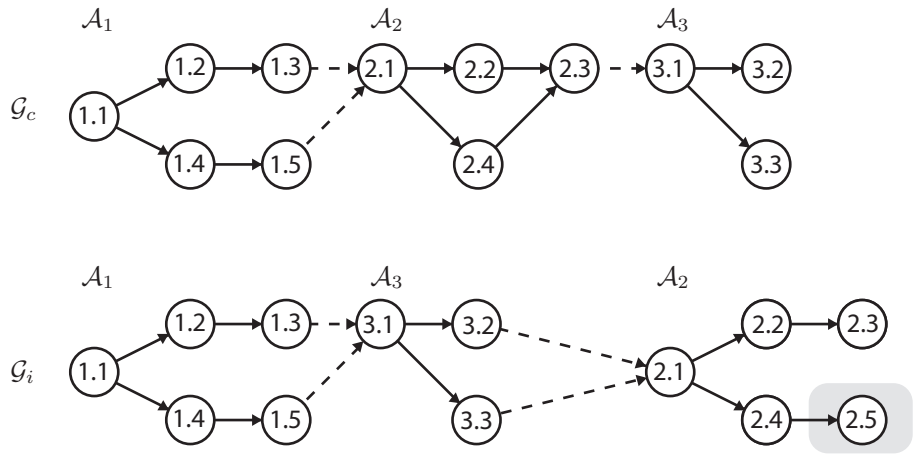

Figure B.5: The scheduler always extracts and then appends the modified application schedule $\mathcal{A}_2$ (which has an additional performer 2.5) to the candidate graph $\mathcal{G}_i$.

then posts it to the conductor, informs the requesting application via its notification conduit that it has successfully posted cndt($\mathcal{A}_k$) and sets $\mathcal{A}_k$ to cndt($\mathcal{A}_k$).

All schedule modifications are serialized through the scheduler thread. Serialization ensures that only one application schedule graph changes in any particular cycle of the schedule life-cycle discussed in section B.4.6. Serializing schedule changes in the scheduler thread also prevents a possible race condition when two or more applications submit candidate schedules in the same basic period.

Figure B.5 shows an example. If application 2 invokes the `restructure` method with the modified arg($\mathcal{A}_2$), then while the current schedule

$$\mathcal{G}_c(i) = \mathcal{A}_1 \oplus \mathcal{A}_2 \oplus \mathcal{A}_3,$$

the newly posted schedule is

$$\mathcal{G}_i = \mathcal{A}_1 \oplus \mathcal{A}_3 \oplus \text{cndt}(\mathcal{A}_2)$$

where $\oplus$ is the schedule assembly operator defined below in section B.4.3.

Always forcing a modified application schedule to the end of the complete schedule graph helps satisfy the principle of modularity. A modified application candidate schedule such as cndt($\mathcal{A}_2$) could contain a never previously executed performer. Such a performer could cause a scheduling failure such as an overtime. Forcing cndt($\mathcal{A}_2$) to the end of the candidate schedule $\mathcal{G}_i$ insures that an overtime in this new performer will not interfere with the execution of the more established performers contributed by other applications.

The `restructure` method signals the status of verifying and posting a schedule arg($\mathcal{A}_k$) with the following return codes.

Listing B.5: Scheduler Interface

```
package com.liquimedia.api;
public interface PrivilegedScheduler extends Scheduler {
    final static int SUCCESS = 0;
    final static int ASYNCH_ALTERATION = -1;

    Application[] getApplicationRanking();
    void setApplicationImportance(Application[] adjusted);
}
```

`BAD_GRAPH`
: The submitted application schedule graph is structurally invalid.

`TOOSLOW`
: The submitted graph is structurally correct but one or more of its paths failed the lifetime admission control test. An application can request a redistribution in response to such an error. In this case, the scheduler continues to use any performers that $\mathcal{A}_k$ contributed to $\mathcal{G}_c$

`POSTED` The scheduler creates, verifies and posts the application candidate schedule and sets $\mathcal{A}_k = \text{cndt}(\mathcal{A}_k)$.

`FAILED` An unspecified error occurs.

Adjusting Importance

The scheduler's privileged interface `PrivilegedScheduler` is the second interface by which an application can modify the scheduling of its performers. It permits an application to adjust the relative importance of its performers. However unlike schedule restructuring, applications cannot directly use this interface to adjust their importance. Instead, an application must request that the user uses an appropriately privileged system utility to adjust application importance. Such a utility uses the `PrivilegedScheduler` shown in listing B.5 to set the importance of all applications in the system.

The `getApplicationRanking` method returns an array of application handles in descending order of importance. The `setApplicationImportance` method instructs the scheduler to reassemble the application schedule graphs in the order specified by `adjusted` and then reverify and post the new schedule graph.

If other applications alter the schedule in between the call to `getApplicationRanking` and `setApplicationImportance`, then `setApplicationImportance` fails and returns `ASYNCH_ALTERATION`. If the scheduler successfully posts the modified schedule, it sends messages to all

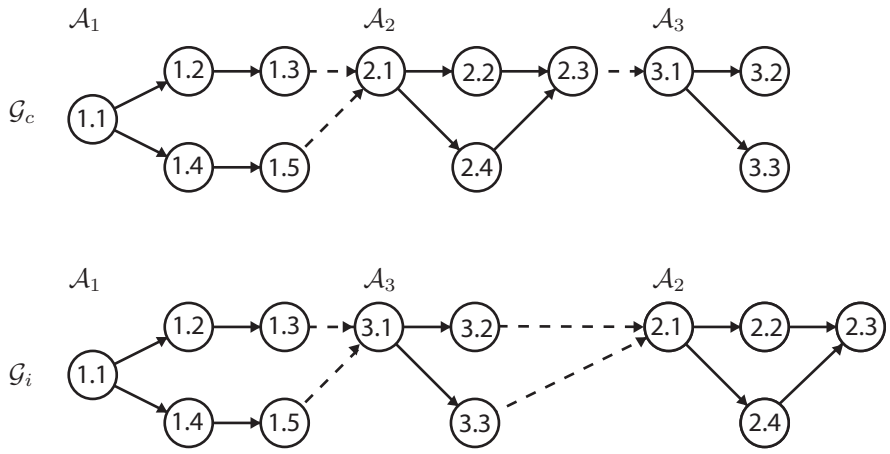

Figure B.6: The scheduler alters the assembly order from $\mathcal{A}_1 \oplus \mathcal{A}_2 \oplus \mathcal{A}_3$ to $\mathcal{A}_1 \oplus \mathcal{A}_3 \oplus \mathcal{A}_2$.

of the affected applications via their message conduits indicating that the user has forcibly adjusted their relative importance.

Figure B.6 shows an example. The call to `getApplicationRanking` returns $[1, 2, 3]$ corresponding to

$$\mathcal{G}_c(i) = \mathcal{A}_1 \oplus \mathcal{A}_2 \oplus \mathcal{A}_3.$$

The user adjusts the relative importance of these three applications with the importance manager utility and applies the change. The importance manager utility then invokes the `setApplicationImportance` method with the argument $[1, 3, 2]$. The scheduler then assembles a candidate schedule

$$\mathcal{G}_i = \mathcal{A}_1 \oplus \mathcal{A}_3 \oplus \mathcal{A}_2$$

verifies it and posts it.

B.4.5 Lifetime Admission Control

LiquiMedia's lifetime admission control mechanism verifies that every path

$$X = \text{path}_i(\text{first}(\text{cndt}(\mathcal{G})); \text{cndt}(\mathcal{G}))$$

in a candidate schedule graph $\text{cndt}(\mathcal{G})$ satisfies

$$P\left((\sum_{p \in X} T_p) < T_B\right) \geq P_c. \tag{B.17}$$

Section B.2.2 describes using Chebyshev's inequality to compute an estimate of a single performer's running time stat(p) with probability P_c. In a similar fashion, Chebyshev's inequality can estimate the running time of multiple performers and admit a schedule if its estimated running time is less than the duration of a basic period.

Assume as a notational convenience that all the performers in cndt($\mathcal{G}$) are simple so that $|\text{path}^*(\text{first}(\text{cndt}(\mathcal{G})); \text{cndt}(\mathcal{G}))| = 1$ and let

$$\mathcal{P} = \text{path}_1(\text{first}(\text{cndt}(\mathcal{G})); \text{cndt}(\mathcal{G})).$$

Then, the scheduler computes

$$\text{stat}(\mathcal{P}) = m_{\mathcal{P},i} + \sqrt{\frac{1}{1-P_c}} s_{\mathcal{P},i} \tag{B.18}$$

where

$$m_{\mathcal{P},i} = w_c + |\mathcal{P}| * w_p + \sum_{\forall p \in \mathcal{P}} m_{p,i} \tag{B.19}$$

$$s_{\mathcal{P},i} = \sqrt{\sum_{\forall p \in \mathcal{P}} s_{p,i}^2} \tag{B.20}$$

and it admits the schedule cndt($\mathcal{G}$) if stat($\mathcal{P}$) $< T_B$.

Equations B.20 and B.20 assume that the PDF of a performer's running time is independent of the path which selected the performer for invocation. Such an assumption is valid because LiquiMedia expects performers to be single-function media-processing functions. Instead of a single monolithic task containing many different behaviours, the schedule graph permits defining an application's range of operations as paths comprised of many single-function performers. LiquiMedia's ability to schedule independently authored performers depends in part on the way that the schedule graph exposes more of an application's structure to both the scheduler and conductor.

Algorithm 6 precisely specifies the operation of the admission control mechanism.

B.4.6 Life Cycle and Feedback

LiquiMedia's scheduler repeats the lifetime admission control test described in section B.4.5 on the running schedule $\mathcal{G}_c$ at intervals proportional to the schedule's age. These additional tests verify that $\mathcal{G}_c$ remains admissible after the inclusion of $M_{p,i,j}$, $S_{p,i,j}$ and $N_{p,i,j}$ values collected by the conductor. If $\mathcal{G}_c$ fails an admission control test, the scheduler creates a new candidate schedule and removes performers from it in order of increasing importance until the candidate is admissible.

Algorithm 6 Lifetime Admission Control

Require: Some candidate schedule $\mathcal{G}$
  $a \leftarrow$ true
  for $\text{path}_i(1;\mathcal{G}) \in \text{path}^*(1;\mathcal{G})$ **do**
    $m \leftarrow w_c$
    $s^2 \leftarrow 0$
    for $p \in \text{path}_i(1;\mathcal{G})$ **do**
      $m \leftarrow m + m_{p,i} + w_p$
      $s^2 \leftarrow s^2 + s^2_{p,i}$
    end for
    $\text{stat}(\text{path}_i(1;\mathcal{G})) \leftarrow m + \sqrt{\frac{1}{1-P_c}}s$
    if $\text{stat}(\text{path}_i(1;\mathcal{G})) \geq T_B$ **then**
      $a \leftarrow$ false
      break
    end if
  end for
  if a is true **then**
    admit schedule $\mathcal{G}$
  end if

The conductor monitors the current scheduler's age and requests a re-admission when the current schedule's age is 10, 35, 105, 4561 and any multiple of 9123 basic periods. These values correspond approximately to .1, .5, 1.5, 60 and 120 seconds. These values were chosen based on the results of Section 6.5.1 and knowledge of a human's 200 millisecond segregation threshold. The first test at 10 basic periods removes unsupportable pre-threshold performers from the schedule. Re-admission tests at 35 and 105 basic periods permit the scheduler to correct the schedule as soon as the collected estimates of a performer's mean are within 1% of an entire sequence's value. Testing at 4561 and repeatedly every 9123 basic periods thereafter incorporates accurate standard deviation values in the tests and watches for changes in performer behaviour. Re-verifications every 9123 basic periods also permits the scheduler to prevent numerical overflow by reseting the values of $M_{p,i,j}$, $S_{p,i,j}$ and $N_{p,i,j}$.

The conductor requests re-admission tests by writing a message to the *re-verification conduit*. It uses a lookup table to determine when the schedule requires a re-verification. It also instructs the scheduler to re-verify the current schedule after any basic period in which the conductor suspends a performer for an error, overtime or exceeding the deferral limit.

Conductor-requested re-verifications have two possible results: the current schedule passes the admission test and the conductor continues using it (albeit with reset $M_{p,i,j}$, $S_{p,i,j}$ and $N_{p,i,j}$ values) or the scheduler posts a new schedule $\mathcal{G}_{i+1}$. Re-verifications are a form of negative feedback as this new schedule $\mathcal{G}_{i+1}$ always contains fewer performers than the schedule $\mathcal{G}_i$ which

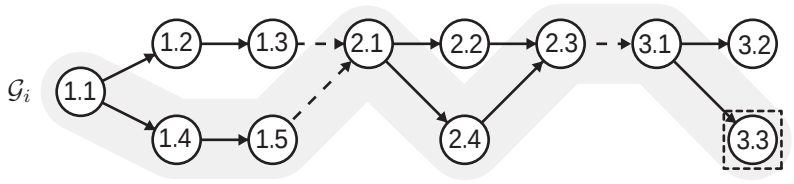

Figure B.7: The diagram outlines a path through the schedule $\mathcal{G}_i$ which is not admissible in gray. The scheduler attempts to make $\mathcal{G}_i$ by removing the boxed performer 3.3.

became the current schedule.

During the construction of any schedule $\mathcal{G}_i$, the scheduler first removes all suspended performers. If $\mathcal{G}_i$ is then inadmissible, the scheduler removes performers from the ends of each failing path in $\text{path}^*(\text{first}(\mathcal{G}_{i+1}); \mathcal{G}_{i+1})$ until all paths pass the lifetime admission control test.

Figure B.7 shows an example of this process. $\text{path}_i(1.1; \mathcal{G}_i)$'s predicted running time exceeds T_B where

$$\text{path}_i(1.1; \mathcal{G}_i) = \{1.1, 1.4, 1.5, 2.1, 2.4, 2.3, 3.1, 3.3\}.$$

Consequently, the scheduler first removes performer 3.3. The scheduler will remove performers from the end of the gray path until it can admit the schedule.

The scheduler's use of negative feedback provides a consistent and stable mechanism for implementing importance-based scheduling. The scheduler packs the schedule with performers in decreasing order of importance. Then, over successive re-verification cycles, the conductor and the scheduler remove the lowest importance performers whose processor resource needs cannot be accommodated.

Appendix C

Windows NT Scheduling Failures

Existing desktop operating systems cannot allocate processor resources in a fashion that permits the successful generation of realtime streams by more than one process at a time. This appendix describes a simple experiment that demonstrates this fact under Windows NT.

C.1 Apparatus

The experimental apparatus consisted of a dual-processor workstation running Windows NT with sufficient memory that paging issues could be ignored. The machine has the WinAmp MP3 audio player program and the QuickTime for Windows image viewer installed. The NT Performance monitor was used to collect results.

C.2 Procedure

The WinAmp MP3 audio player successfully generated a music stream when it was the only application generating a media stream. The human auditory system can monitor a PCM audio stream for missing samples and can easily detect as few as three missed samples. Any samples missing from the output stream show a scheduling failure where the audio player cannot execute before the hardware audio buffer drains. Consequently, it was easy to hear that WinAmp decompressed the audio in realtime when it was the only multimedia application running.

Once the WinAmp audio stream was successfully established, I opened multiple images with the QuickTime image viewer and counted the approximate number and duration of glitches in the audio stream. The chosen images caused the image viewer to use a CODEC intended for

decompression of DV digital video. Using the image viewer was a necessary expedient because, while a video player would have been a preferable second process, the available video players block at startup waiting for WinAmp to close the audio device.

C.3 Results

The digital audio player used an average of 3.2% of a processor to perform decompression. The audio stream is deeply buffered — hardware information for the machine indicates that the audio is fed from an 8092 byte hardware buffer, giving a minimum of 46 milliseconds of audio buffering before an audible glitch. This is more than three basic periods in LiquiMedia.

The image CODEC is single-threaded and uses exactly 100% of one processor. Windowing operations such as the creation of windows to contain the decompressed images and transfer the pixels to the screen were carried out by the second processor and consumed less than 30% of its capacity.

The audio glitched reliably, with pauses approaching an appreciable fraction of a second, if the decompressor was started with at least 24 images.

The above describes a situation in which Windows NT fails to schedule two separate stream-generating processes despite adequate processing resources. Consequently, Windows NT cannot successfully schedule all combinations of multimedia streams even when sufficient processor resources are available.

Appendix D

LiquiMedia Inc.

This thesis may have somewhat of a "commercial" bias: the introduction motivates the research by its practical applications and the discussion of previous research includes numerous products in addition to references to the academic literature. My attempt at starting a company to commercialize the LiquiMedia technology explains this bias. This appendix provides a brief overview of LiquiMedia Inc. (LMI) and summarizes its accomplishments.

D.1 The Company

I founded LMI in cooperation with the Eastern Technology Seed Investment Fund (ETSIF). ETSIF financed the company under the mentor capital model where ETSIF provided LMI with both seed capital and interim management assistance.

ETSIF insisted that the company be founded before I completed the thesis in order to insure that the public disclosure of thesis submission did not jeopardize a patent application on the LiquiMedia technology. In retrospect, this was an error. At first, developing the `liqui` prototype benefited both the thesis and the company. However, company responsibilities such as patent and white paper writing soon supplanted thesis work.

It was my understanding that the company's interim management would carry out five fundamental tasks. First, the interim management established the company's "plumbing" — systems such a accounting and disbursement control. Second, the interim management was responsible for securing the company's intellectual property. In particular, LMI applied for a patent on the LiquiMedia architecture (United State Patent Application 09/259,178).

Third, the interim management was responsible for establishing a business strategy that could generate revenue with a LiquiMedia-derived product. This task was a pre-requisite for

the two final tasks: using the revenue potential of the chosen business strategy to obtain a subsequent round of financing and hiring full-time management. The company's interim management failed to perform these tasks. With insolvency imminent, I left LMI to return to work on this thesis.[1]

D.2 Research

I produced all of LMI's technical literature. This included a sizable number of technical white papers describing both the LiquiMedia architecture and both the advantages and defects of related commercial developments. Chapter 3 discusses these products as well as results from the academic literature.

Working at LMI helped me develop an appreciation for the business utility of a technology such as LiquiMedia. While I began this thesis with nothing more than the desire to build novel multimedia software, such an attitude is not useful in a commercial setting. An infrastructural technology such as LiquiMedia has commercial value if its use improves the bottom line. The processor coalescence example described in Section 1.3.2 has this potential.

D.3 Developments

LMI used the code tree developed for this thesis to implement a realtime-capable Java virtual machine. Figure D.1 shows the general architecture. As discussed in Chapter 5, the `liqui` prototype is built on top of Solaris. LMI integrated Sun's Java virtual machine (JVM) into the `liqui` prototype.

LMI's development had two main components. First, it developed the "RTX" realtime extension that tied together a binary-only Java virtual machine with the existing `liqui` prototype. This extension enabled Java composers to create and install Java performers into the LiquiMedia scheduler and for these code fragments to use the conduit facility. As part of this effort, LMI also identified a restricted subset of Java that performers could successfully use.

Second, LMI developed a audio player application in Java that would demonstrate that LMI's combination of LiquiMedia, Solaris and the Java virtual machine could in fact execute a mixture of realtime and non-realtime fragments written in Java. The demonstrator was a simulation of a portable MP3 player, such as the Diamond RiO [Dia00]. The Java Media Framework provided non-realtime decoding of the compressed audio. Running the JMF's decoder as a performer would have required access to the both the JVM and JMF sources and would have required development effort which exceeded the available time. Java performers provided a latency-controlled graphic equalizer and software volume control.

[1] The compactness of this paragraph belies the unpleasantness.

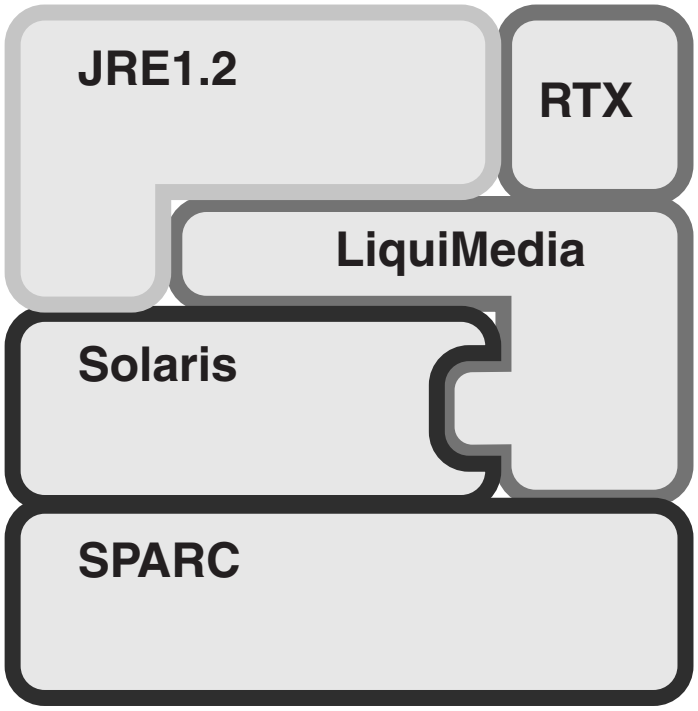

Figure D.1: LiquiMedia Inc. Demonstrator

This combination of non-realtime decoding and realtime post-processing demonstrated that LiquiMedia could successfully partition the operation of realtime and non-realtime tasks. It also demonstrated the operational correctness of the conduit library. Performance data from the audio processing performers showed that multimedia performers have running time distributions similar to the Pareto distribution where statistical scheduling offers a sizable utilization advantage over rate monotonic scheduling.

D.3.1 Audioplayer Lessons

The implementation of the audio player provided a number of valuable practical lessons: the difficulty of realtime development and the particular difficulty of realtime development without complete control of the infrastructure. This section discusses these lessons briefly.

First, the version of the Sun Java virtual machine used by the LMI development team may choose to perform a garbage collection on any attempt to allocate storage. This includes the allocation of a stack frame for a Java method invocation. The execution of all Java code is blocked while the garbage collector runs. Garbage collection pauses typically range from 30ms to 150ms. Consequently, Java performers in the LMI realtime JVM must minimize their memory allocations. Insuring that collections did not interfere with Java performer execution was the foremost difficulty in obtaining realtime execution of Java code in the LMI JVM.

Second, Solaris 2.6 does not implement priority inheritance in a single process containing realtime and non-realtime kernel threads. This complicated the conductor's support for reentrant operation.

Third, the Sun audio device driver uses Streams [Bac86]. The keyboard and mouse drivers also use Streams. Interlocks inside the stream stack can deadlock the kernel when realtime and non-realtime threads in kernel context contest for access to in-kernel stream functionality.

Finally, LiquiMedia's statistical admission control and full support for reentrant operation in the conductor were invaluable for bringing realtime to Java. The combination of Java's garbage collection and its inter-operation with the remainder of Solaris make the JVM a realtime-uncooperative environment. Java performers have exceedingly long tails in their running time distributions. Statistical scheduling's tolerance for such "messy" distributions was essential to successfully execute Java performers in realtime.

D.4 Summary

Working at LMI was extremely educational. It provided an opportunity to evaluate the thesis research against criteria atypical for academic work: its potential for commercial application. The LiquiMedia technology has sizable commercial applicability — it can lower overall system

costs for devices that must execute several multimedia tasks. However, commercializing a body of research requires management, marketing and business development skills that computer science coursework failed to provide.

Appendix E

PDF Statistical Scheduling

Appendix B presented a detailed description of how LiquiMedia uses Chebyshev's inequality to compute its estimate of a performer's running time stat(p) with at least some minimum probability P_c. Use of Chebyshev's inequality permits computing stat(p) for an arbitrary performer.

Chebyshev's inequality does not provide the tightest possible bound stat(p). Estimating stat(p) using p's true probability distribution function (PDF) $\phi_p(t)$ provides the smallest possible value of stat(p). However, $\phi_p(t)$ cannot be determined. This appendix discusses PDF-style estimation: the use of a function $\hat{\phi}_p(t)$ to approximate the unknown $\phi_p(t)$. It also shows that the normal distribution is a poor choice for $\hat{\phi}_p(t)$.

E.1 PDF-style Estimation

Assume that there exists some PDF $\hat{\phi}_p(t)$ which when parametrized with p's (full-sequence) mean μ_p and standard deviation σ_p satisfies equation E.1 with at least P_g probability.

$$P(\hat{\phi}_p(t; \mu_p, \sigma_p) \approx \phi_p(t)) \leq P_g \tag{E.1}$$

Then, stat(p) is the solution of equation E.2.

$$\int_{t=0}^{t=\text{stat}(p)} \hat{\phi}_p(t; \mu_p, \sigma_p) dt = \frac{P_g}{P_c} \tag{E.2}$$

Solving equation E.2 for stat(p) would produce a much smaller value of stat(p) than the one estimated by equation B.5 if equation E.1 holds with probability $P_g > P_c$. Consequently, PDF-style estimation requires a two step process: the scheduler first verifies that equation E.1 holds and then solves equation E.2.

Verifying equation E.1 is a classical problem in statistics: test the hypothesis that $\hat{\phi}(t; \mu_p, \sigma_p)$ fits p's actual running times $t_{p,1}, \ldots t_{p,i}$ for i invocations of performer p. I chose the rank scores technique for this verification as it permits using linear regression to evaluate how well a non-linear function fits a sample [WM78].

Embedding this verification process into LiquiMedia complicates the implementation as the conductor must collect and preserve a sample window of per-invocation running times for each performer. Further, the scheduler must include code to compute a linear regression on the sample versus its rank scores. This hypothesis verification step alone has a considerably larger memory footprint and scheduler execution overhead than the entire resource use of the Chebyshev-style estimator. Consequently, its impact on the principle of ultra-fine granularity is only justified if its per-performer utilization benefits exceed the increased scheduler overhead.

Having completed the first verification step, the scheduler must actually compute a value of stat(p). Given an integrable function $\hat{\phi}_p(t)$ such that

$$\hat{\Phi}_p(t) = \int \hat{\phi}_p(t)dt$$

then equation E.2 can be re-expressed (note that the probability of performer execution times ≤ 0 is 0) as shown in equation E.3.

$$\text{stat}(p) = \hat{\Phi}^{-1}_{p;\mu_p,\sigma_p}(\frac{P_g}{P_c}) \tag{E.3}$$

The scheduler repeatedly computes the values of stat(p) for hundreds or even thousands of performers and so the choice of $\hat{\phi}_p(t)$ must permit the scheduler to efficiently compute equation E.3.

As with the use of Chebyshev's inequality, the values of μ_p and σ_p are not available. The function $\hat{\phi}_p(t)$ must instead be parametrized with a sample mean and standard deviation corrected as discussed in section B.2.2.

E.2 Normal Distribution

Successful PDF-style estimation requires a choice of function for $\hat{\phi}_p(t)$. The normal distribution warranted investigation as a possible choice for two reasons. First, as required by the central limit theorem, the distribution of the combined running times of many performers (such as a single path through the schedule graph) will tend to be normally distributed as the number of performers increase [WM78]. Second, it is easy to numerically solve equation E.3 for stat(p) when using the normal distribution as the function $\hat{\phi}_p(t)$.

However, the normal is a poor estimator of the standard test performers discussed in Chapter 6 and will fare even worse for realistic performers as they have large positive skew and resemble a Pareto distribution.

Running Time	Simple	Synth	Jittered	Sinusoidal
.01	.325	.278	.572	.742
.05	.523	.695	.852	.952
.10	.641	.763	.964	.975
.20	.774	.868	.971	.975
.40	.871	.894	.992	.966
.75	.667	.943	.980	.955

Table E.1: Normal Score Correlations

I evaluated the normal distribution's ability to estimate a performer's running time by producing normal score plots for the standard test performers. The correlation between the n normal scores $\nu_1, \ldots, \nu_n$ and the actual performer runtime data $t_{p,1} \ldots t_{p,n}$ where

$$\nu_i = \text{Gau}^{-1}(\frac{i - 1/3}{n + 1/3})$$

and Gau^{-1} is the inverse normal cumulative distribution function provides a numerical measure of how closely a normal distribution approximates the running time of a performer p. If the sample distribution is perfectly normal, the normal score would be 1.0.

Table E.1 summarizes the correlation between the performers' normal scores and their actual running times. While an appropriate distribution to estimate a performer's running time might improve processor utilization, the normal distribution is not a sufficiently good estimator of performer run time to warrant its use. For example, for .40 Jittered, the performer with the most normally distributed run time distribution, at $P_c = .99$, the normal provides only a 8% tighter bound than using Chebyshev's inequality.

In the case of .01 Simple, the normal estimator provides a 20% tighter stat(p) than the Chebyshev-style estimator. However, with a correlation between its normal scores and sample distribution of .325, the tighter bound is specious.

Obviously the normal distribution is not a good choice for $\hat{\phi}_p(t)$. Many other possible distributions exist. For example, the Pareto distribution ought to be a better choice [AB98b]. Regardless of the chosen distribution's ability to fit the actual performer invocation running times, the greater complexity of the PDF-style estimator's two-phase verification process favours the use of the Chebyshev-style estimator. Further, the two-phase verification process may unfairly reject performers whose running time distribution does not match the chosen approximating PDF. Consequently, use of PDF-style estimation is only warranted when the operating system domain requires maintaining high utilization levels at values of P_c above those needed by multimedia applications.

Appendix F

Testing Summary

F.1 Introduction

F.1.1 Test Performers

Almost all of the tests had the following general structure:

1. a test harness composer and
2. some number of performers drawn from the standard test performers discussed in Section 6.1.3.

F.2 Unit Testing

The following unit tests were applied to their corresponding components.

`cord_test.c`
: String handling [Boe04].

`storage_test.c`
: Instrumented storage and interfaces to garbage collector [Boe04].

`closures_test.c`
: Thread wrappers for terminating composer threads.

`composer_test.c`
: Composer task creation and cleanup.

`conduit_test.c`
: Conduit asynchronous message passing facilities.

`group_test.c`
: Schedule graph structures. Schedule graphs are also re-used to store composer task information in `liqui`.

`impresario_test.c`
: Loading of application shared libraries, initialization, composer task creation and clean-up.

`lead_test.c`
: Schedule path management.

`performer_test.c`
: Performer management data structures, creation, clean-up, statistical profile collection and computation.

`properties_test.c`
: Schedule graph annotations.

`schedule_test.c`
: Scheduler, admission testing and schedule restructuring.

`signalhandlers_test.c`
: Exception handling and portions of priority inheritance.

F.3 Integration Testing

Implementation tests are pass or fail. Each test applied a stimulus to `liqui` while its state met the test's specified pre-condition. To pass, `liqui` satisfied the post-condition after the application of the stimulus. Some of the tests require `liqui` to correctly reach the post conditions within a bounded amount of time. Test descriptions use the notation explained in Appendix B.

1. App `liqui_harness`, single spinning composer, debugger.
   Pre A composer executing without a lock.
   Stim A single basic period interrupt sent to the conductor.
   Post One normal conductor iteration.
2. App `liqui_harness`, single spinning composer, debugger.
   Pre A composer executing with a locked conduit mutex.
   Stim A single basic period interrupt sent to the conductor.
   Post The conductor blocked on the conduit mutex.

3. App `liqui_harness`, single spinning composer, debugger.
   Pre A performer executing.
   Stim A single basic period interrupt sent to the conductor.
   Post Performer suspension code (overtime case) executed (c.f. iii).
4. App `liqui_harness`, single spinning composer, debugger.
   Pre Conductor running but not in the context of a performer.
   Stim A single basic period interrupt sent to the conductor.
   Post Conductor restarts without handling overtime but checks for need to increment progress values (c.f. i, ii, iv).
5. App `empty_startup.c`
   Pre A ready to run `liqui` without an application.
   Stim Start `liqui` for a small fixed number of cycles.
   Post Message log contains all service composer start and stop messages in the correct order.
6. App `chatty_performer.c`
   Pre A running `liqui` with heavy-weight logging and no performers.
   Stim `app_main` creates a new group containing a single correct performer and installs it. The performer logs its invocations.
   Post Message log contains all performer invocation messages.
7. App `chatty_performer.c`
   Pre A running `liqui` with no installed performers.
   Stim `app_main` schedules a performer which logs its age and profile.
   Post Logged ages and statistical profiles are correct.
8. App `seg_fault_performer.c`
   Pre One installed performer p.
   Stim p generates an exception in basic period i.
   Post p is suspended for an error and has no light-weight statistical profile records for basic periods $> i$.
9. App `basic_performer.c`
   Pre One installed performer p.
   Stim p enters an infinite loop in basic period i.

Post At the end of conductor initialization in basic period $i+1$, p is suspended (overtime) and has no light-weight records for basic periods $\geq i$.

10. App `multi_performer.c`

    Pre Installed performers $1, \ldots, p_l$.

    Stim Randomly chosen $p \in [1, p_l)$ enters an infinite loop in basic period i.

    Post p is suspended due to overtime, $\text{prgs}(x) = 1; x \in [1, p-1]$, $\text{prgs}(y) = 2; y \in [p+1, p_l]$

11. App `arbitrary_delta.c`

    Pre Installed performer p having $\text{prgs}(p) \neq 1$ in basic period $i-1$.

    Stim p executes successfully in basic period i.

    Post $\text{prgs}(p) = 1$ after completion of execution of performer p in basic period i.

12. App `domino.c`

    Pre Installed performers $1, \ldots, p_l$ with p_l greater than the maximum number of successive deferrals.

    Stim $p \ldots p+n$ enter infinite loops in basic periods $i \ldots i+n$ respectively

    Post At the end of basic period $i+n$, all performers $p, \ldots, p+n$ are suspended for overtime and all performers $p+n, \ldots, p_l$ are suspended for too many successive deferrals.

13. App `ot1_test.c`

    Pre Installed performers $1 \ldots p_l$.

    Stim In basic period i, the aggregate running time of performers $1, \ldots, p-1$ exceeds $\text{lt}(p, P_c)$.

    Post At the end of basic period i, p is deferred and has $\text{prgs}(p) = 2$.

14. App `restructuring_tests.c`

    Pre For k admitted applications with candidate schedule contributions: $\mathcal{A}_1, \ldots, \mathcal{A}_k$ such that

$$\mathcal{G}_c = \bigoplus_{\kappa=1}^{k} \mathcal{A}_\kappa.$$

    Stim Application i invokes `schedRestructureSchedule` [1].

    Post Posted schedule $\mathcal{G}_i$ is

$$\mathcal{G}_i = \bigoplus_{\kappa=1}^{i-1} \mathcal{A}_\kappa \oplus \bigoplus_{\kappa=i+1}^{k} \mathcal{A}_\kappa \oplus \mathcal{A}_i$$

[1] The C function equivalent to `restructure` in Section B.4.4.

15. App `restructuring_tests.c`

    Pre For k admitted applications with candidate schedule contributions: $\mathcal{A}_1, \ldots, \mathcal{A}_k$ with performer p in some application candidate schedule $\mathcal{A}_\kappa$

    Stim In basic period i, p enters an infinite loop.

    Post Posted schedule $\mathcal{G}_i$ does not contain performer p and application κ receives an overtime suspension notice in its notification conduit.

16. App `restructuring_tests.c`

    Pre For k admitted applications with candidate schedule contributions: $\mathcal{A}_1, \ldots, \mathcal{A}_k$ such that

    $$\mathcal{G}_c = \bigoplus_{\kappa=1}^{k} \mathcal{A}_\kappa.$$

    and un-scheduled but admissible application candidate $\mathcal{A}_{k+1}$.

    Stim Application $k+1$ invokes `schedRestructureSchedule`.

    Post Posted schedule $\mathcal{G}_i$ is

    $$\mathcal{G}_i = \bigoplus_{\kappa=1}^{k+1} \mathcal{A}_\kappa$$

Appendix G

Approximating wcet(p)

Chapter 6 makes the following approximation:

$$\text{wcet}(p) \approx \max_{i=1,\ldots,n} t_{p,i}$$

because it is impossible to actually compute $\max_{i=1,\ldots,n} t_{p,i}$ without knowing the PDF of performer p.

It is possible to quantify the probability that this approximation places the Chebyshev estimator at a disadvantage compared to the WCET estimator. For n random variables $X_1, \ldots, X_n$ selected from the PDF $f_p(x)$ of performer p, then

$$\begin{aligned} P(\forall i \in [1,n]; X_i < \max(x)) &= \prod_{i=1}^{n} P(X_i < \max(x)) && \text{(G.1)} \\ &= \prod_{i=1}^{n} \int_0^{\max(x)} f_p(x)dx && \text{(G.2)} \\ &= \left(\int_0^{\max(x)} f_p(x)dx \right)^n && \text{(G.3)} \end{aligned}$$

But in the sample of $n = 4561$ random variables,

$$\begin{aligned} P(\forall i \in [1,n]; X_i < \max(x)) &= \frac{4560}{4561} && \text{(G.4)} \\ &\approx 0.9998 && \text{(G.5)} \end{aligned}$$

so

$$\begin{aligned} \int_0^{\max(x)} f_p(x)dx &= \left(\frac{4560}{4561} \right)^{1/4561} && \text{(G.6)} \\ &\approx 0.99999995. && \text{(G.7)} \end{aligned}$$

Consequently, given that no value of X can exceed the length of the basic period, the probability that there is a X larger than $\max(x)$ is extremely small.