Operating System Support for Multimedia: The Programming Model Matters

John Regehr, Michael B. Jones[†], John A. Stankovic

September 2000

Technical Report
MSR-TR-2000-89

Department of Computer Science
Thornton Hall, University of Virginia
Charlottesville, VA 22903-2242, USA
john@regehr.org, stankovic@cs.virginia.edu
`http://www.cs.virginia.edu/holst`

[†] *Microsoft Research, Microsoft Corporation*
One Microsoft Way, Building 112/2156
Redmond, WA 98052, USA
mbj@microsoft.com
`http://research.microsoft.com/~mbj`

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://research.microsoft.com`

Operating System Support for Multimedia:
The Programming Model Matters

John Regehr, Michael B. Jones[†], John A. Stankovic

Department of Computer Science
Thornton Hall, University of Virginia
Charlottesville, VA 22903-2242, USA
john@regehr.org, stankovic@cs.virginia.edu
`http://www.cs.virginia.edu/holst`

[†]*Microsoft Research, Microsoft Corporation*
One Microsoft Way, Building 112/2156
Redmond, WA 98052, USA
mbj@microsoft.com
`http://research.microsoft.com/~mbj`

September 2000

Abstract

Multimedia is an increasingly important part of the mix of applications that users run on personal computers and workstations. The requirements placed on a multimedia operating system are demanding and often conflicting: untrusted, independently written soft real-time applications must be able to coexist without interfering with each other. This must be accomplished while requiring as little extra effort as possible from application developers, and the resulting system must be usable and understandable by end users even when application resource requirements exceed system capacity.

This article analyzes the goals of multimedia schedulers and provides a taxonomy of techniques used to achieve them; representative schedulers are classified and characterized in terms of the things that they make easy and hard, including the associated programming tasks. This is done to support our principal contribution: an analysis of usability issues and tradeoffs in multimedia scheduling for both application developers and end users.

1 Introduction

Multimedia is an increasingly important part of the mix of applications that users run on personal computers and workstations. A large subset of multimedia applications (that are sometimes called *continuous media*) require soft real-time service from the operating system.

The set of abstractions and conventions implemented by a particular system that allow soft real-time applications to meet their requirements defines a *programming model*. A multimedia scheduler must support a programming model that is useful and understandable to the people who develop applications for the system. Furthermore, the scheduler, in conjunction with applications, must meet user expectations and provide understandable behavior in the face of sets of applications that demand resources exceeding the capacity of the system.

Many systems supporting multimedia and other soft real-time applications have been described, each typically claiming unique advantages that make it better than the others that came before. Yet, curiously, often the very features claimed as advantages by one system are instead characterized as disadvantages by the next.

For example, SMART [17] and Rialto [11] both offer deadline-based scheduling to applications in the form of *time constraints*. To use a time constraint, an application requests an amount of CPU time before a deadline (20ms of processing during the next 100ms, for example); the scheduler then notifies the application that the constraint is either accepted or rejected.

Although both systems provide the same basic abstraction, the guarantees they offer are different. Once Rialto informs an application that a requested time constraint is feasible it guarantees that the time for that constraint will have been dedicated to the requesting application no matter what other applications might do. SMART, on the other hand, may invalidate a previously accepted time constraint part way through its execution, taking away its reserved time, if a higher-priority application requests a conflicting time constraint (in which case the original application will receive an exception). So, SMART potentially provides faster response time for higher-priority applications that unexpectedly request CPU time, but Rialto supports a programming model in which a developer does not need to worry about the case in which a feasible time constraint is not actually scheduled.

In both cases the designers of the system believed they were making the right decision. How can it be that one man's features are another man's bugs? Clearly the authors have not agreed upon either the goals they were trying to achieve or the criteria by which their systems should be judged. Tradeoffs such as the one in this example are important because they affect the basic set of assumptions that programmers can make while implementing multimedia applications.

This article analyzes the sets of goals that multimedia schedulers might try to achieve. It creates a taxonomy of the kinds of programming models used to achieve these goals and it characterizes a number of representative systems used to run multimedia applications within this taxonomy.

It is our intent for the use of this taxonomy to enable: (i) careful comparisons to be made between existing work, (ii) the identification of new parts of the design space leading to possible new solutions, and (iii) a better understanding of how the needs of several types of multimedia applications are served by (or are not well-served by) the various programming models promoted by important types of multimedia schedulers. All of this is done to support our principal contribution: an analysis of usability issues and tradeoffs in multimedia scheduling for both application developers and end users.

2 Multimedia System Requirements

2.1 Context for the Requirements

A *general-purpose operating system* (GPOS) for a PC or workstation must provide fast response time for interactive applications, high throughput for batch applications, and some amount of fairness between applications. Although there is tension between these requirements, the lack of meaningful changes to the design of time-sharing schedulers in recent years would seem to indicate that they are working well enough.

The goal of a hard real-time system is similarly unambiguous: all hard deadlines must be met. The design of the system is dictated by this requirement, although it conflicts to some extent with designing a low-cost system. Despite the conflict, there appears to be a standard engineering practice for building such systems: statically determine resource requirements and then overprovision processor cycles as a hedge against unforeseen situations.

Not surprisingly, there is a large space of systems whose requirements fall between these two extremes. These are soft real-time systems: they need to support a dynamic mix of applications, some of which must perform computations at specific times. Missed deadlines are undesirable, but not catastrophic. In this article we are concerned with the requirements placed upon multimedia operating systems, which are usually, but not necessarily, general-purpose operating systems with soft real-time extensions.

We have attempted to identify an uncontroversial set of requirements that the "ideal" multimedia operating system would meet. Although it is unlikely that any single system or scheduling policy will be able to meet all of these requirements for all types of applications, the requirements are important because they describe the space within which multimedia systems are designed. A particular set of prioritizations among the requirements will result in a specific set of tradeoffs, and these tradeoffs will constrain the design of the user interface and the application programming model of a system.

While our focus is on CPU scheduling, we recognize that other resources can be just as important to the timely and efficient execution of multimedia applications. The principles behind our technique of analyzing and evaluating programming models also apply to the algorithms that allocate other resources.

2.2 List of Requirements

R1: *Meet the scheduling requirements of coexisting, independently written, possibly misbehaving soft real-time applications.* The CPU requirements of a real-time application are often specified in terms of an *amount* and *period*, where the application must receive the amount of CPU time during each period of time. No matter how scheduling requirements are specified, the scheduler must be able to meet them without the benefit of global coordination among application developers—multimedia operating systems are *open systems* in the sense that applications are written independently.

A misbehaving application (from the point of view of the scheduler) will *overrun* by attempting to use more CPU time than was allocated to it. Schedulers that provide *load isolation* guarantee a minimum amount or proportion of CPU time to each multimedia application even if other applications overrun (by entering an infinite loop, for example).

R2: *Minimize development effort by providing abstractions and guarantees that are a good match for applications' requirements.* An important role of the designers of soft real-time systems is to ease application developers into a world where their application gracefully shares machine resources with other applications. We propose the following test: compare the difficulty of writing an application for a given multimedia scheduler to the difficulty of writing the same application if it could assume that it is the highest priority application in the system (thus having the machine logically to itself). If the difference in costs is too high, application developers will assume that contention does not exist. Rather than using features provided by the scheduler, they will force their users to manually eliminate contention.

Getting the programming model right is very important: if a system becomes widely used, the effort expended by application developers will far outweigh the effort required to implement the system. It is therefore possible for small increases in usability to justify even large amounts of implementation complexity and effort. In other words, the programming model matters.

R3: *Provide a consistent, intuitive user interface.* Users should be able to easily express their preferences to the system and the system should behave predictably in response to user actions. Also, it should give the user (or software operating on the user's behalf) feedback about the resource usage of existing applications and, when applicable, the likely effects of future actions.

R4: *Run a mix of applications that maximizes overall value.* Unlike hard real-time systems, PCs and workstations cannot overprovision the CPU resource; demanding multimedia applications tend to use all available cycles. During overload, the multimedia OS should run a mix of applications that maximizes overall value. Value is a subjective measure of the utility of an application, running at a particular time, to a particular user.

3 Basis of a Taxonomy

The top level of our taxonomy of scheduling support for multimedia applications distinguishes between steady-state allocation of CPU time and system behavior during application mode changes (when an application starts, terminates, or has a change of requirements). In both parts of the taxonomy key questions are:

- What information do applications have to provide the system with in order to use the programming model?
- What kinds of guarantees does the system make to applications?
- What kinds of applications does the programming model support well (and poorly)?
- Whose jobs does it make easier (and harder)?
- How comprehensible and usable is the resulting programming interface?
- How comprehensible and usable is the resulting user interface?

3.1 Steady State Allocation of CPU Time

For each scheduler, we provide a brief description, give examples of systems that implement it, and examine which of the requirements from Section 2 the scheduler fulfills. These characteristics are summarized in Table 1.

programming model	examples	load isolation	prior knowledge	support for varying latency requirements?
rate-monotonic and other static priority	Linux, RTLinux, Solaris, Windows 2000	isolated from lower priority	priority	yes
CPU reservations	Nemesis, Rialto, Spring	strong	period, amount	yes
proportional share	BVT, EEVDF, SMART	strong	share (, latency)	varies
earliest deadline first	Rialto, SMART	strong / weak	amount, deadline	yes
feedback control	FC-EDF, SWiFT	varies	metric, set point	varies
hierarchical scheduling	CPU Inheritance, SFQ	varies	varies	varies

Table 1: Characterization of soft real-time schedulers.

3.1.1 Static Priority and Rate Monotonic Scheduling

The uniprocessor real-time scheduling problem has essentially been solved by *static priority analysis* [1] when the set of applications and their periods and amounts are known in advance, and when applications can be trusted not to overrun. Static-priority analysis is a generalization of *rate-monotonic analysis* [14].

Popular general-purpose operating systems such as Linux and Windows 2000 extend their time-sharing schedulers to support static priority threads that have strictly higher priority than any time-sharing thread. Schedulers with this structure exhibit well-known pathologies such as unbounded priority inversion (unless synchronization primitives have been augmented to support priority inheritance) and starvation of time-sharing applications during overload [16]. Furthermore, developers are likely to overestimate the priority at which their applications should run because a poorly performing application reflects negatively on its author. This phenomenon is known as *priority inflation*.

Although static priority schedulers are simple, efficient, and well understood, they fail to isolate applications from one another, and optimal priority assignment requires coordination among application developers. Applications can only be guaranteed to receive a certain amount of CPU time if the worst-case execution times of higher-priority applications are known, and this is generally not possible. Still, the static-priority programming model is reasonably intuitive for both users (if an application is starving, there must be overload at higher priorities) and programmers (higher priority applications run first), and it supports legacy applications.

3.1.2 CPU Reservations

A *CPU reservation* provides an application with load isolation and periodic execution. For example, a task could reserve 10ms of CPU time out of every 50ms; it would then be guaranteed to receive no less than the reserved amount per period.

The original Spring kernel [20] is an example that represents one end of the reservation spectrum, i.e., it provides precise hard real-time guarantees. To achieve these hard guarantees Spring required significant amounts of a priori information and associated tools to extract that information. For example, the Spring programming language had restrictions placed on it such as capping all loops, no dynamic memory management, etc. Due to the cost of runtime support in this system, this solution is not suitable for continuous media. However, the Spring system was then extended in [12] to integrate continuous multimedia streams into this hard guarantee paradigm.

In general-purpose operating systems reservations can be implemented in a variety of ways. Nemesis [13] uses an earliest deadline first (EDF) scheduler in conjunction with an enforcement mechanism; a rate-monotonic scheduler could also be used. Rialto [11] uses a tree-based data structure to represent time intervals.

CPU reservations satisfy the requirement of supporting coexisting, possibly misbehaving real-time applications. They eliminate the need for global coordination because application resource requirements are stated in *absolute* units (time) rather than *relative* units like priority or share. However, reservation-based schedulers must be told applications' periods and amounts. The period is easier to determine: the characteristics of a periodic thread, such as its data rates, buffer sizes, and latency requirements typically dictate its period; likewise, applications often implicitly make it available to the operating system by using a timer to awaken a thread every time the period begins. The amount of CPU time can be difficult to predict, as it is both platform and data dependent. For some applications a good estimate of future amount can be obtained by averaging previous amounts; other applications such as the notoriously variable MPEG video decoder inherently show wide fluctuations in amount [2]. Underestimates of amounts will sometimes prevent application requirements from being met, and overestimates will result in needless rejection of multimedia applications. Furthermore, determining CPU requirements through measurement begs the question of how to tell when a program is behaving normally and when it is overrunning.

Because reservations provide applications with fairly hard performance guarantees (how hard depends on the particular implementation), they are best suited for scheduling applications that lose much of their value when their CPU requirements are not met. Reservations can be used to support legacy multimedia applications if the period and amount can be determined from outside the applications and applied to them without requiring modifications.

3.1.3 Proportional Share

Proportional share schedulers are quantum-based weighted round-robin schedulers that guarantee that an application with N shares will be given at least N/T of the processor time, on average, where T is the total number of shares over all applications. This means that the absolute fraction of the CPU allocated to each application decreases as the total number of shares increases, unless the system recomputes shares. Quantum size is chosen to provide a good balance between allocation error and system overhead.

Other than Lottery scheduling [23], which is a randomized algorithm, all proportional share algorithms appear to be based on a *virtual clock*—a per-application counter that the scheduler increments in proportion to the amount of real time the application executes and in inverse proportion to the application's share. At each reschedule, the scheduler dispatches the runnable application with the lowest virtual clock value.

Some proportional share algorithms decouple an application's share from its latency requirement—this is a critical property for real-time schedulers. EEVDF [22] achieves this by allowing clients to individually make the tradeoff between allocation accuracy and scheduling overhead. SMART [17] supports a mixed programming model in which applications receiving proportional share scheduling can meet real-time requirements using the deadline-based *time constraint* abstraction. BVT [6] associates a *warp* value with each application; positive warp values allow a thread to build up credit while blocked, increasing the chances that it will be scheduled when it wakes up. (Nemesis provides a *latency hint* that is similar to warp: it brings the effective deadline of an unblocking thread closer, making it more likely to be scheduled.) The *hybrid lottery scheduler* described by Petrou et al. [19] automatically provides improved response time for interactive applications.

Without admission control, proportional share schedulers will not be able to guarantee that any particular application will receive even its minimum CPU requirement during overload. Therefore, proportional share schedulers best support applications that *degrade gracefully*, or lose value smoothly and in proportion to the amount of CPU time taken away from them. For example, in response to a shortage of cycles a game or other real-time renderer can reduce its frame rate; an MPEG video player can reduce its frame rate and drop frames. Other applications do not gracefully degrade: software modems and audio players lose most of their value if they receive even slightly less CPU time than their full requirement.

3.1.4 Earliest Deadline First

EDF is an attractive scheduling discipline because it is optimal in the sense that if there exists any algorithm that can schedule a set of tasks without missing any deadlines, then EDF can also schedule the tasks without missing any deadlines. Soft real-time OSs primarily use EDF to keep track of deadline urgency inside the scheduler; only a few systems such as Rialto and SMART have exposed deadline-based scheduling abstractions to application programmers. Both systems couple deadlines with an admission test (because EDF does not work well during overload) and call the resulting abstraction a *time constraint*.

Time constraints present a fairly difficult programming model because they require fine-grained effort: the application programmer must decide which pieces of code to execute within the context of a time constraint in addition to providing the deadline and an estimate of the required processing time. Applications must also be prepared to skip part of their processing if the admission test fails. Once a time constraint is accepted, Rialto guarantees the application that it will receive the required CPU time. SMART will sometimes deliver an upcall to applications informing them that a deadline previously thought to be feasible has become infeasible.

3.1.5 Feedback-Based Scheduling

Multimedia OSs need to work in situations where total load is difficult to predict and execution times of individual applications vary considerably. To address these problems new approaches based on feedback control have been developed. Feedback control concepts can be applied at admission control and/or as the scheduling algorithm itself.

In the FC-EDF work [15] a feedback controller is used to dynamically adjust CPU utilization in such a manner as to meet a specific set point stated as a deadline miss percentage. FC-EDF is not designed to prevent individual applications from missing their deadlines; rather, it aims for high utilization and low overall deadline miss ratio.

SWiFT [21] uses a feedback mechanism to estimate the amount of CPU time to reserve for applications that are structured as pipelines. The scheduler monitors the status of buffer queues between stages of the pipeline; it attempts to keep queues half full by adjusting the amount of processor time that each stage receives.

Both SWiFT and FC-EDF have the advantage of not requiring estimates of the amount of processing time that applications will need. Both systems require periodic monitoring of the metric that the feedback controller acts on.

3.1.6 Hierarchical Scheduling

Hierarchical (or multi-level) schedulers generalize the traditional role of schedulers (i.e., scheduling threads or processes) by allowing them to allocate CPU time to other schedulers. The *root* scheduler gives CPU time to a scheduler below it in the hierarchy and so on until a leaf of the scheduling tree—a thread—is reached.

The scheduling hierarchy may either be fixed at system build time or dynamically constructed at run time. *CPU inheritance scheduling* [7] probably represents an endpoint on the static vs. dynamic axis: it allows arbitrary user-level threads to act as schedulers by *donating* the CPU to other threads.

Hierarchical scheduling has two important properties. First, it permits multiple programming models to be supported simultaneously, potentially enabling support for applications with diverse requirements. Second, it allows properties that schedulers usually provide to threads to be recursively applied to groups of threads. For example, a fair-share scheduler at the root of the scheduling hierarchy on a multi-user machine with a time-sharing scheduler below it for each user provides load isolation between users that is independent of the number of runnable threads each user has. A single-level time-sharing or fair-share scheduler does not do this.

Hierarchical Start-Time Fair Queuing (SFQ) [8] provides flexible isolation using a hierarchical proportional share scheduler. Deng et al. [5] describe a two-level scheduling hierarchy for Windows NT that has an EDF scheduler at the root of the hierarchy and an appropriate scheduler (rate-monotonic, EDF, etc.) for each real-time application.

3.2 System Behavior During Mode Changes

We characterize system behavior during application mode changes by looking at the various kinds of guarantees that the operating system gives applications. The guarantee is an important part of the programming model since it determines what assumptions the programmer can make about the allocation of processor time that an application will receive.

When the OS gives an application a guarantee, it is restricting its future decision making in proportion to the strength of the guarantee. Seen in this light, it is understandable that many systems give applications weak or nonexistent guarantees—there is an inherent tradeoff between providing guarantees and dynamically optimizing value by allocating cycles on the fly in response to unexpected demand.

3.2.1 Best Effort

Best effort systems make no guarantees to applications. Rather than rejecting an application during overload, a best effort system reduces the processor time available to other applications to “make room” for the new one. This works well when application performance degrades gracefully.

Although “best effort” often has a negative connotation, it does not need to imply poor service. Rather, a best-effort system avoids the possibility of needlessly rejecting feasible applications by placing the burden of avoiding overload on the user. The computer and user form a feedback loop, where the user manually reduces system load after observing that applications are performing poorly.

We propose two requirements that applications must meet for “feedback through the user” to work. First, applications must degrade gracefully. Second, application performance must not be hidden from the user, who has to be able to notice degraded performance in order to do something about it. An application that fails both of these criteria is the software controlling a CD burner: it does not degrade gracefully since even a single buffer underrun will ruin a disc, and the user has no way to notice that the burner is running out of buffers supplied by the application.

3.2.2 Admission Control

A system that implements *admission control* keeps track of some metric of system load, and rejects new applications when load is above a threshold. For systems implementing reservations, system load could be the sum of the processor utilizations of existing reservations.

Because it can be used to prevent overload, admission control allows a multimedia system to meet the requirements of all admitted applications. It provides a simple programming model: applications are guaranteed to receive the amount of resources that they require until they terminate or are terminated (assuming that CPU requirements can be accurately estimated at the time a program first requests real-time service). Admission control also makes the system designer’s job easy: all that is required is a load metric and a threshold.

Admission control does not serve the user well in the sense that there is no reason to believe that the most recently started application is the one that should be rejected. However, when a valuable application is denied admission the user can manually decrease the load on the system and then attempt to restart the application. Obviously, this feedback loop can fail when the admission controller rejects a job not directly initiated by the user (for example, recording a television show to disk while the user is not at home).

3.2.3 Resource Management: System Support for Renegotiation

Best effort and admission control are simple heuristics for achieving high overall value in situations where the user can take corrective action when the heuristic is not performing well. *Resource management* techniques attempt to achieve high overall value with little or no user intervention. They do this by stipulating that guarantees made to applications may be renegotiated to reflect changing conditions. Renegotiation is initiated when the resource manager calculates that there is a way to allocate CPU time that is different from current allocations that would provide higher value to the user. To perform this calculation, the system must have, for each application, some representation of the relationship between the resources granted to the application and the application’s perceived value to the user.

Oparah [18] describes a resource management system that extends Nemesis; it has the interesting feature that the user can assign positive or negative feedback to decisions made by the resource manager. This is designed to bring the resource manager’s actions more closely into line with user preferences over time.

3.2.4 Adaptation: Application Support for Renegotiation

Adaptive applications support different modes of operation along one or more dimensions. For example, a video player may support several resolutions, frame-rates, and compression methods. Each mode has a set of resource requirements and offers some value to the user. The promise of adaptive applications is that a resource manager will be able to select modes for the running set of applications that provide higher overall value than would have been possible if each application had to be either accepted at its full service rate or rejected outright.

The *imprecise computation model* [9] permits fine-grained adaptation by dividing computations into mandatory and optional parts, where the optional part adds value, if performed.

Assuming that an application already supports different modes, adaptation complicates the application programming model only slightly, by requiring the application to provide the system with a list of supported modes and to change modes asynchronously in response to requests from the system. Adaptive systems also require a more careful specification of what guarantees are being given to applications. For example, is an application asked if it can tolerate degraded service, is it told that it must, or does it simply receive less processor time without being notified? Is renegotiation assumed to be infrequent, or might it happen often?

Adaptation does not appear to make the user's job, the programmer's job, or the system designer's job any easier. Rather, it permits the system to provide more value to the user. A possible drawback of adapting applications is that users will not appreciate the resulting artifacts, such as windows changing size and soundtracks flipping back and forth between stereo and mono. Clearly there is a cost associated with each user-visible adaptation; successful systems will need to take this into account.

3.3 Practical Considerations

Programming models encompass more than high-level abstractions and APIs: any feature (or misfeature) of an operating system that the programmer must understand in order to write effective programs becomes part of the programming model. In this section we explore a few examples of this.

Can applications that block expect to meet their deadlines? Analysis of blocking and synchronization is expected for hard real-time systems; soft real-time programs are usually assumed to not block for long enough to miss their deadlines. Applications that block on calls to servers can only expect the server to complete work on their behalf in a timely way if the operating system propagates the client's scheduling properties to the server, and if the server internally schedules requests accordingly.

Does dispatch latency meet application requirements? Dispatch latency is the time between when a thread is scheduled and when it actually runs. It can be caused by the scheduling algorithm or by other factors; for example, in a GPOS a variety of events such as interrupt handling and network protocol processing can delay thread scheduling. Non-preemptive operating systems exacerbate the problem: a high priority thread that wakes up while the kernel is in the middle of a long system call on the behalf of another thread will not be scheduled until the system call completes. Properly configured Windows 2000 [4] and Linux machines have observed worst-case dispatch latencies[1] below 10ms—this meets the latency requirements of virtually all multimedia applications. Unfortunately, their real-time performance is fragile in the sense that it can be broken by any code running in kernel mode. Device drivers are particularly problematic; rigorous testing of driver code is needed in order to reduce the likelihood of latency problems [10]. Hard real-time operating systems keep interrupt latencies very low and prohibit other kinds of unscheduled CPU time; they may have worst-case thread dispatch latencies in the tens of microseconds.

Is the same programming model available to all threads? Very low dispatch latency can be achieved using co-resident operating systems [3]. This approach virtualizes the interrupt controller seen by a general-purpose operating system in order to allow a small real-time kernel to run even when the GPOS has "disabled interrupts." The GPOS runs in the idle time of the real-time kernel; the two OSs may then communicate through FIFOs that are non-blocking from the real-time side. The programming environment presented by the real-time kernel is sparse (since it cannot easily invoke services provided by the GPOS) and unforgiving (mistakes can easily hang or crash the machine). However, this is a useful approach for highly latency-sensitive applications that can be divided into real-time and non-real-time components.

[1] Based on dispatch latency measurements while the system is heavily loaded. This is not a true worst-case analysis but it indicates that the systems can perform well in practice.

type	examples	period	amount	degrades gracefully?	latency sensitivity
stored audio	MP3, AAC	around 100ms	1%–10%	no	low
stored video	MPEG-2, AVI	33ms	large	yes	low
distributed audio	Internet telephone	bursty	1%–10%	no	high
distributed video	video conferencing	bursty	large	yes	high
real-time audio	software synthesizer	1–20ms	varies	no	very high
RT simulation	virtual reality, Quake	up to refresh period	usually 100%	yes	high
RT hardware	soft modem, USB speakers	3–20ms	up to 50%	no	very high

Table 2: Characterization of soft real-time applications.

4 Applications Characterized

The real-time requirements imposed on an operating system are driven by the applications that must be supported. This section briefly describes the main characteristics of several important categories of applications; these are summarized in Table 2.

Applications that play stored audio and video are characterized by the lack of a tight end-to-end latency requirement: large buffers of encoded and decoded data can be stored in order to tolerate variations in disk, network, and processor bandwidth. The only latency-sensitive part of the video display process is switching the frame that is being displayed. The latency sensitivity of a digital audio player is determined by the size of the buffer on the sound hardware. Video players can degrade gracefully by dropping frames; audio players are not able to do this and will cause annoying sound glitches if their CPU requirements are not met. Although decoding audio streams in formats such as MP3 and AAC (MPEG-2 Advanced Audio Coding—a compressed audio format that is similar to MP3) does not require a substantial fraction of a modern CPU, decoding video can be CPU intensive, especially when the display adapter does not provide hardware acceleration. Encoding MPEG-2 streams in software is much more CPU-intensive than decoding them; high-quality real-time encoding is just becoming possible.

For other applications, latency sensitivity comes from a timing dependency between a data source and sink. For example, video frames received by a telepresence or video conferencing application must be displayed shortly after they are received—the requirement for low perceived latency precludes deep buffering. Audio and video applications, live or recorded, can, in principle, be adaptive. However, current applications tend to either not be adaptive, or to be manually adaptive at a coarse granularity. For example, although Winamp, a popular MP3 player, can be manually configured to reduce its CPU usage by disabling stereo sound, it has no mechanism for doing this in response to a shortage of processor cycles.

When a computer is used as a real time audio mixer or synthesizer, the delay between when a sound arrives from a peripheral and when it is played must not exceed about 20ms if the sound is to be perceived as simultaneous with the act of playing it. Reliable fine-grained (small millisecond) real-time is barely within reach of modern general-purpose OSs. Real-time audio synthesis is especially demanding because, in some cases, it is closer to hard real-time than soft: during a recording session the cost of a dropped sample may be large.

The rendering loop in immersive 3D environments and games such as Doom and Quake must display frames that depend on user input with as little delay as possible in order to be convincing and avoid inducing motion sickness. Rendering loops are usually adaptive, using extra CPU cycles to provide as many frames per second as possible, up to the screen refresh rate. Consequently, these applications are almost always CPU bound and they cannot easily share the processor with other applications, unless the scheduler can enforce a limit on the CPU usage of the game.

Finally, the high average-case performance of modern processors and low profit margins in the PC industry create powerful incentives for peripheral designers to push functionality from hardware into software. For example, software modems contain a bare minimum of hardware, performing all signal processing tasks in software on the main CPU(s). This requires code to be reliably scheduled every 3-16ms; missed deadlines reduce throughput and may cause dropped connections. USB speakers and CD burners also require real-time response from the OS in order to avoid sound glitches and ruined discs, respectively.

A trend opposing the migration of functionality into software is the decreasing size and cost of embedded computers; this makes it inexpensive to perform real-time tasks on special-purpose hardware instead of on general-purpose operating systems. However, the low cost of downloading a software module (compared to acquiring a new embedded device) ensures that users will want to perform real-time tasks on PCs during the foreseeable future. Furthermore, we believe that PCs will continue to have abundant resources compared to special-purpose devices (although PCs often lack dedicated hardware that enables some kinds of tasks to be performed much more easily).

5 Challenges for Practical Soft Real-Time Scheduling

In Section 2 we presented several requirements that a good multimedia OS should fulfill; in this section we refocus those requirements into a set of research challenges for future systems.

C1: *Create user-centric systems.* Users tell the system how to provide high value—they start up a set of applications and expect them to work. Resource management systems should respect a user's preferences when tradeoffs need to be made between applications, and should seek to maximize the utility of the system as perceived by the user. User studies are needed in order to figure out how admission control and adaptation can be used in ways that are intuitive and minimally inconvenient to users.

C2: *Create usable programming models.* In addition to the usual questions about how effective, novel, and efficient a scheduler is, we believe that the systems research community should be asking:

- What assumptions does it make about application characteristics, and are these assumptions justified?
- Can application developers use the programming model that is supported by the proposed system? Is it making their job easier?
- Are applications being given meaningful guarantees by the system?

C3: *Design schedulers and metrics that are robust with respect to unpredictability.* Traditional real-time analysis assumes that software execution times can be predicted. Unfortunately, a number of hardware and software trends are making predictability an increasingly difficult goal. These trends include deeper caching hierarchies, increasing prevalence of multiprocessors (and eventually, multi-threaded processors), variable processor speeds for power and heat management, larger and more deeply layered software bases, and just-in-time translation, optimization, and virtualization of binaries. Increasing unpredictability means that we need scheduling techniques that are more adaptive, where both applications and the system monitor and react to application progress. We also need metrics for soft real-time (traditional metrics such as number of missed deadlines are no longer sufficient) that provide a means with which to talk and reason about the complex relationship between scheduling unpredictability and loss of value in applications.

C4: *Provide scheduling support for applications with diverse requirements.* We believe that multimedia systems should support at least three types of scheduling: (i) guaranteed rate and granularity scheduling for real-time applications that do not degrade gracefully, (ii) best-effort real-time scheduling for real-time applications that degrade gracefully, and (iii) time-sharing support for non-real-time applications.

C5: *Provide integrated scheduling of all important resources.* Although we have concentrated on CPU scheduling in this article, other resources such as disk, network, and memory also need to be scheduled in order to achieve overall application predictability. Not only must these resources be scheduled, but we also need to understand the interactions between policies scheduling various resources. Finally, scheduling information must be propagated between subsystems in order to support end-to-end guarantees.

6 Conclusions

Scheduling support for multimedia does not exist in a vacuum: schedulers only make sense within the context of a set of requirements of applications and, ultimately, users. This article has provided a framework by which the differing goals of many of the multimedia schedulers in research and production operating systems might be compared and evaluated. Rather than making value judgments about one system being better than another in an absolute sense, we have characterized each in terms of the different things that they make easy and hard, including the associated programming tasks.

As in the realm of programming languages, there are probably multiple "sweet spots" in operating system support for multimedia applications. It is our hope that this article will aid the research community in constructively comparing their systems in this space, and indeed, to help find these "sweet spots" and promote the construction of systems filling them.

Acknowledgments

The authors would like to thank Tarek Abdelzaher, David Coppit, Kevin Jeffay, Chenyang Lu, Stefan Saroiu, Leigh Stoller, and Kevin Sullivan for their helpful comments on drafts of this article.

References

[1] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.

[2] Andy C. Bavier, A. Brady Montz, and Larry L. Peterson. Predicting MPEG execution times. In *Proc. of the Joint International Conf. on Measurement and Modeling of Computer Systems*, pages 131–140, Madison, WI, June 1998.

[3] Gregory Bollella and Kevin Jeffay. Support For Real-Time Computing Within General Purpose Operating Systems: Supporting co-resident operating systems. In *Proc. of the 1st IEEE Real-Time Technology and Applications Symposium*, pages 4–14, Chicago, IL, May 1995.

[4] Erik Cota-Robles and James P. Held. A Comparison of Windows Driver Model Latency Performance on Windows NT and Windows 98. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, pages 159–172, New Orleans, LA, February 1999.

[5] Zhong Deng, Jane W.-S. Liu, Lynn Zhang, Seri Mouna, and Alban Frei. An Open Environment for Real-Time Applications. *Real-Time Systems Journal*, 16(2/3):165–185, May 1999.

[6] Kenneth J. Duda and David C. Cheriton. Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[7] Bryan Ford and Sai Susarla. CPU Inheritance Scheduling. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, pages 91–105, Seattle, WA, October 1996.

[8] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, pages 107–121, Seattle, WA, October 1996.

[9] David Hull, Wu-chun Feng, and Jane W.-S. Liu. Operating System Support for Imprecise Computation. In *Proc. of the AAAI Fall Symposium on Flexible Computation*, pages 9–11, Cambridge, MA, November 1996.

[10] Michael B. Jones and John Regehr. The Problems You're Having May Not Be the Problems You Think You're Having: Results from a Latency Study of Windows NT. In *Proc. of the 7th Workshop on Hot Topics in Operating Systems*, pages 96–101, March 1999.

[11] Michael B. Jones, Daniela Roşu, and Marcel-Cătălin Roşu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 198–211, Saint-Malô, France, October 1997.

[12] Hiroyuki Kaneko, John A. Stankovic, Subhabrata Sen, and Krithi Ramamritham. Integrated Scheduling of Multimedia and Hard Real-Time Tasks. In *Proc. of the 17th IEEE Real-Time Systems Symposium*, Washington, DC, December 1996.

[13] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[14] 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, January 1973.

[15] Chenyang Lu, Jack A. Stankovic, Gang Tao, and Sang H. Son. The Design and Evaluation of a Feedback Control EDF Scheduling Algorithm. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, Phoenix, AZ, December 1999.

[16] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. In *Proc. of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.

[17] Jason Nieh and Monica S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malô, France, October 1997.

[18] Don Oparah. A Framework for Adaptive Resource Management in a Multimedia Operating System. In *Proc. of the 6th IEEE International Conf. on Multimedia Computing and Systems*, Florence, Italy, June 1999.

[19] David Petrou, John W. Milford, and Garth A. Gibson. Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers. In *Proc. of the USENIX 1999 Annual Technical Conf.*, pages 1–14, Monterey, CA, June 1999.

[20] John A. Stankovic and Krithi Ramamritham. The Spring kernel: a new paradigm for real-time systems. *IEEE Software*, 8(3):62–72, May 1991.

[21] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.

[22] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A Proportional Share Resource Allocation Algorithm For Real-Time, Time-Shared Systems. In *Proc. of the 17th IEEE Real-Time Systems Symposium*, pages 288–299, Washington, DC, December 1996.

[23] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proc. of the 1st Symposium on Operating Systems Design and Implementation*, pages 1–11. USENIX Association, 1994.