Operating System Support for Multimedia: The Programming Model Matters

John Regehr
School of Computing
University of Utah
regehr@cs.utah.edu

Michael B. Jones
Microsoft Research
Microsoft Corporation
mbj@microsoft.com

John A. Stankovic
Department of Computer Science
University of Virginia
stankovic@cs.virginia.edu

Abstract

Multimedia is an increasingly important part of the mix of applications that users run on personal computers and workstations. Research in operating system support for multimedia has traditionally been evaluated using metrics such as fairness, the ability to permit applications to meet real-time deadlines, and run-time efficiency. We argue, on the other hand, that if advanced scheduling and resource management techniques are to be adopted in mainstream operating systems, the roles of developers and users will have to be considered much more seriously than they have been up to this point. Our goals are to recast the dialogue about multimedia scheduling in these terms and to inform the reader about the state of the art in this area. To accomplish this we survey existing multimedia scheduling techniques and analyze them in terms of things that they make easy and difficult for whom, including the associated programming tasks.

1 Introduction

Personal computers running Windows XP, MacOS X, and Linux are capable of performing a variety of multimedia tasks — accurately recognizing continuous speech, encoding captured television signals and storing them on disk, acting as professional-quality electronic musical instruments, and rendering convincing virtual worlds — all in real time. Furthermore, personal computers costing less than $1000 are capable of performing several of these tasks at once if the operating system manages resources well. The increasing pervasiveness of multimedia applications, and problems supporting them on traditional systems, have motivated many research papers over the past decade.

This article serves two purposes. First, it surveys existing processor scheduling techniques for supporting multimedia on general-purpose operating systems. Second, it advances and supports the thesis that traditional technical metrics for evaluating these systems, while important, are not going to decide whether or not these techniques are successful in the long run through adoption in mainstream operating systems. Traditional metrics, one or more of which can be found in almost every research paper on the subject, include run-time efficiency, fairness, ability to permit tasks to meet deadlines, and, more recently, energy efficiency. We argue, on the other hand, that the critical metrics are understandability, predictability, and ease of use, not only for application developers but also for end users. As systems software developers we recognize that these metrics are difficult to evaluate. However, we believe that improvements can be made simply by framing the dialogue about multimedia support in these terms. For example, can developers reasonably be expected to provide the information that the system needs? Will users be confronted with new and confusing policy choices to make?

To be widely accepted a multimedia technology must present each of three different groups — operating system vendors, application developers, and end users — with a good value proposition. The rest of this article defends the thesis that *the programming model matters*, where a programming model is the set of abstractions and functionality presented by a system to application developers.

2 Multimedia System Requirements

A *general-purpose operating system* (GPOS) for a personal computer 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 indicates that they are working well enough.

The goal of a hard real-time system is similarly unambiguous: all hard deadlines must be met. The standard engineering practice for building these systems is to statically determine resource requirements and schedulability, as well as over-provisioning resources as a hedge against unforeseen situations.

Not surprisingly, there are many systems whose re-

quirements 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.

We have identified four basic requirements that the "ideal" multimedia operating system should 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; these tradeoffs will constrain the design of the user interface and the application programming model.

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.

From the point of view of the scheduler a misbehaving application 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, e.g. by entering an infinite loop.

R2: *Minimize development effort by providing abstractions and guarantees that are a good match for applications' requirements.*

In the past, personal computers were dedicated to a single application at a time. Developers did not need to interact much with OS resource allocation policies. This is no longer the case. For example, it is possible to listen to music while playing a game, burn a CD while watching a movie, or encode video from a capture card while using speech recognition software. Therefore, an important role of the designers of soft real-time systems is to make it as easy as possible for developers to create applications that gracefully share 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 — reducing the value potentially available to end users.

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. This is the "holy grail" of resource management and is probably impossible in practice since value is a subjective measure of the utility of an application, running at a particular time, to a particular user. Still, this requirement is a useful one since it provides a basis for evaluating different systems.

3 Multimedia Scheduling Strategies

Our survey 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 survey key questions are:

- What information do applications have to provide to the system in order to use the programming model?
- What 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 XP	isolated from lower priority	priority	yes
proportional share	BVT, EEVDF, SMART	strong	share (and latency)	varies
CPU reservations	Nemesis, Rialto, Spring	strong	period, amount	yes
earliest deadline first	Rialto, SMART, Spring	strong / weak	deadline, amount	yes
feedback control	FC-EDF, SWiFT	varies	metric, set point	varies
hierarchical scheduling	CPU Inheritance, SFQ, HLS	varies	varies	varies

Table 1: Characterization of soft real-time schedulers.

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.

Popular general-purpose operating systems such as Linux and Windows XP extend their time-sharing schedulers to support real-time threads that have strictly higher priority than any time-sharing thread. When used in an open system, schedulers with this structure exhibit well-known pathologies such as starvation of time-sharing applications during overload [11] and unbounded priority inversion, unless synchronization primitives have been augmented to support priority inheritance. 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 harmful 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).

Proportional Share

Proportional share schedulers are quantum-based weighted round-robin schedulers. They 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. Quantum size is chosen to provide a good balance between allocation error and system overhead.

Other than Lottery scheduling [18], which is a randomized algorithm, all proportional share algorithms are based on some sort of *virtual clock* — a per-thread counter that the scheduler increments in proportion to the amount of CPU time received by the thread and in inverse proportion to the thread's share. At the start of each quantum the scheduler dispatches the runnable thread 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 [16] achieves this by allowing clients to individually make the tradeoff between allocation accuracy and scheduling overhead. SMART [12] supports a mixed programming model in which applications receiving proportional share scheduling can meet real-time requirements using a deadline-based *time constraint* abstraction. BVT [4] 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 [8] 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.

Without admission control, proportional share schedulers cannot guarantee that an application will receive even its minimum CPU requirement during overload. Proportional share schedulers therefore 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. Other applications do not gracefully degrade: software modems and audio players lose most or all of their value if they receive even slightly less CPU time than their full requirement.

CPU Reservations

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

The original Spring kernel [14] is an example that represents one end of the reservation spectrum, i.e., it provides precise hard real-time guarantees using a scheduler based on the earliest deadline first (EDF) algorithm. To achieve these hard guarantees Spring required significant amounts of a priori information and associated tools to extract that information. Due to the cost of runtime support this solution is not suitable for continuous media. However, the Spring system was later extended 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 uses an EDF scheduler in conjunction with an enforcement mechanism, Rialto [7] uses a tree-based data structure to represent time intervals, and TimeSys Linux/CPU [17] uses a priority-based scheduler.

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 required 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. The period is easier to determine: typically it is not data or hardware dependent, but rather is determined by latency requirements and the sizes of data buffers.

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.

Earliest Deadline First

Earliest deadline first (EDF) is an attractive scheduling discipline because it is theoretically optimal in the sense that, under certain assumptions, if any scheduling algorithm can meet all deadlines then EDF can. Soft real-time OSs primarily use EDF as an internal scheduler implementation technique where it has no impact on the programming model: only a few systems such as Rialto and SMART expose deadline-based scheduling abstractions to application programmers. Both systems couple deadline-based scheduling with an admission test and call the resulting abstraction a *time constraint*.

Time constraints present a fairly difficult programming model because they require fine-grained effort: the developer 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, on the other hand, will sometimes deliver an upcall to applications informing them that a deadline previously thought to be feasible has become infeasible, forcing the program to take appropriate action.

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 [10] 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 [15] 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.

Hierarchical Scheduling

Hierarchical 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* [5] 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. This useful property cannot be provided by a traditional single-level time-sharing or proportional share scheduler.

Hierarchical Start-Time Fair Queuing (SFQ) [6] provides flexible isolation using a hierarchical proportional share scheduler. Deng et al. [3] 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. Regehr and Stankovic [13] developed HLS; its contribution was to permit more effective reasoning about the guarantees provided by heterogeneous hierarchies of schedulers.

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 strong guarantees and dynamically optimizing value by allocating cycles on the fly in response to unexpected demand.

Best Effort

Best effort systems make no guarantees at all. 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 necessarily 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. The software controlling a CD burner fails both of these criteria: it does not degrade gracefully since even a single buffer underrun will ruin a disc, and the user usually has no way to notice that the burner is running out of buffers supplied by the application.

Admission Control

A system that implements *admission control* keeps track of some metric of system load, rejecting 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. 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.

Renegotiation of Guarantees

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. Techniques using *renegotiation* attempt to achieve high overall value with little or no user intervention by stipulating that guarantees made to applications may be modified in response to changes in system load. Renegotiation is initiated when the system calculates that there is a way to allocate CPU time that is different from current allocations that would provide higher value. To perform this calculation the system must have a representation of the relationship between resources granted to applications and applications' perceived value to the user. For example, Li and Nahrstedt [9] describe a framework that provides global coordination among applications and uses feedback from individual applications to determine how they are performing.

Application Adaptation

Adaptation is the application-level counterpart to renegotiation of guarantees, where an adaptive application supports 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 the system 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.

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 when requested. 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 make the user's job, the programmer's job, or the system designer's job any easier. Instead, 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 must take this cost 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 only if the operating system propagates the client's scheduling properties to the server, and if the server internally schedules requests accordingly. Servers for the X Window System running on UNIX-like operating systems are a good example where neither requirement is typically met: this is a continuing source of trouble for UNIX-based multimedia applications.

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. Operating systems with non-preemptible kernels 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 NT [2] and Linux machines have observed worst-case dispatch latencies[1] below 10 ms — this meets the latency requirements of virtually all multimedia applications. Unfortunately, the real-time performance of these systems is highly 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.

Hard real-time operating systems keep interrupt latencies very low by exercising rigid control over code that executes in kernel mode; they may have worst-case thread dispatch latencies in the tens of microseconds. General-purpose operating systems have tended to slowly chip away at latency problems by fixing trouble spots. Recently there have been a number of versions of Linux that provide enhanced real-time characteristics to applications; they have accomplished this by making the kernel preemptible and by breaking up long critical

[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.

sections in the kernel.

4 Characterizing Applications

The real-time requirements imposed on an operating system are driven by applications. 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 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 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; real-time encoding with good compression ratios 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 10–20 ms 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 limit 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. This requires code to be reliably scheduled every 3-16 ms; 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.

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 are made between applications and should seek to maximize the utility of the system as perceived by the user. User studies are needed to figure out how admission control and adaptation can be used in ways that are intuitive and minimally inconvenient.

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

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

C3: *Design schedulers and metrics that are robust with respect to unpredictability.* Traditional real-time

type	examples	period	amount	degrades gracefully?	latency sensitivity
stored audio	MP3, AAC	around 100 ms	1%–5%	no	low
stored video	MPEG-2, AVI	33 ms	large	yes	low
distributed audio	Internet telephone	bursty	1%–5%	no	high
distributed video	video conferencing	bursty	large	yes	high
real-time audio	software synthesizer	1–20 ms	varies	no	very high
RT simulation	virtual reality, Quake	up to refresh period	usually 100%	yes	high
RT hardware	soft modem, USB speakers	3–20 ms	up to 50%	no	very high

Table 2: Characterization of soft real-time applications.

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 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. These metrics will provide the means for talking and reasoning 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: guaranteed rate and granularity scheduling for real-time applications that do not degrade gracefully, best-effort real-time scheduling for real-time applications that degrade gracefully, and 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 and control the interactions between policies scheduling various resources.

6 Conclusions

Scheduling support for multimedia does not exist in a vacuum: schedulers only make sense within the context of the requirements of applications, developers, and, ultimately, users. This article has evaluated the differing goals of many of the multimedia schedulers in research and production operating systems. 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] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, Sept. 1993.

[2] E. Cota-Robles and J. P. Held. A comparison of Windows Driver Model latency performance on Windows NT and Windows 98. In *Proc. of the 3rd Symp. on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.

[3] Z. Deng, J. W.-S. Liu, L. Zhang, S. Mouna, and A. Frei. An open environment for real-time applications. *Real-Time Systems Journal*, 16(2/3):165–185, May 1999.

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

[5] B. Ford and S. Susarla. CPU inheritance scheduling. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*, pages 91–105, Seattle, WA, Oct. 1996.

[6] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*, pages 107–121, Seattle, WA, Oct. 1996.

[7] M. B. Jones, D. Roşu, and M.-C. Roşu. CPU Reservations and Time Constraints: Efficient, predictable scheduling of independent activities. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 198–211, Saint-Malô, France, Oct. 1997.

[8] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. 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, Sept. 1996.

[9] B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptation. *IEEE Journal on selected Areas of Communication*, 17(9):1632–1650, Sept. 1999.

[10] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. The design and evaluation of a feedback control EDF scheduling algorithm. In *Proc. of the 20th IEEE Real-Time Systems Symp.*, pages 56–67, Phoenix, AZ, Dec. 1999.

[11] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall. SVR4 UNIX scheduler unacceptable for multimedia applications. In *Proc. of the 4th Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, Nov. 1993.

[12] J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, Saint-Malô, France, Oct. 1997.

[13] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proc. of the 22nd IEEE Real-Time Systems Symp.*, pages 3–14, London, UK, Dec. 2001.

[14] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–72, May 1991.

[15] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proc. of the 3rd Symp. on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.

[16] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proc. of the 17th IEEE Real-Time Systems Symp.*, pages 288–299, Washington DC, Dec. 1996.

[17] TimeSys Linux/GPL. `http://www.timesys.com/prodserv`.

[18] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation*, pages 1–11, 1994.