LiquiMedia — A dynamically extensible cyclic executive

Robert Kroeger, William Cowan

November 21, 1996

Because it involves theoretical latencies that are unbounded, and practical latencies that are large, dynamic resource allocation poses difficult problems for all real-time systems. Control applications, where the task environment and range of possible computations are both highly constrained, can solve this problem using static allocation, with all allocation occurring outside of the real-time constraints. This simple and reliable strategy is unsuited to multimedia systems, however, because they must operate successfully in extremely variable computational environments.

The solution is to create a version of dynamic allocation that retains the essence of the static solution, while taking advantage of characteristics unique to multimedia. This paper discusses using this strategy for allocation of computation and communication. CPU cycles and network bandwidth are the most basic resources used in any computation. We believe that a clean abstraction that is successful with these resources will certainly extend to other resources such as input and output devices.

We believe that current operating systems designed for multimedia cannot allocate computational resources at a sufficiently small granularity. Current general purpose operating systems allocate a processor's time with threads. Even on the fastest systems, context switches result in delays of tens of microseconds. As a result, using the processor efficiently requires task durations a millisecond or longer. Common tasks for a multimedia operating system such as synthesizing positionalized sound or animating sprites take far less than a millisecond for each iteration. Consequently, a multimedia operating system must provide an alternative to threads for the allocation of processor time.

We call our alternative *performers*. Performers discard the overhead of context switching by abandonning preemption. They must always yield the processor voluntarily. Forcing programmers to statically divide the application into groups of performers makes the same tradeoff as a cyclic executive: additional development work in exchange for no runtime resource allocation overhead.

This tradeoff engenders a new problem. Unlike a traditional fixed-functionality cyclic executive, a multimedia operating system needs the capability of dynamically altering which performers it's executing. Maintaining realtime execution of performers requires that the operating system attempt to execute only per-

formers which can complete their execution within a specified interval. Here lies the fundamental resource allocation problem underlying a cyclic executive style of multimedia operating system: how to determine the elapsed duration of a randomly selected group of performers before actually executing them.

Before presenting our solution to this resource allocation problem, we will further justify our rejection of threads. Obviously, context switching between threads can take significant time on conventional architectures. We obtained the following experimental results for context switch times for realtime threads running on various SPARC processors under Solaris2.5.[1]

Processor	*Average delay*	*Lower bound*
UltraSparc, 140MHz	17μs	4μs
MicroSparc2, 110MHz.	52μs	8μs
SuperSparc, 50MHz	29μs	4μs
SparcStation 1, 20MHz	500μs	14μs

Table 1: Context Switch Times

While this does not represent the apex of realtime responsiveness, it does reflect average context switch times in a state of the art general purpose operating system. The above table also contains an estimate of the context switch delay. This shows that while an operating system specifically designed for realtime will do better, it cannot achieve more than a factor of four improvement when running on contemporary hardware. While hardware architecture changes may provide hardware-assisted context switching,[2] making efficient use of current hardware under a threaded multimedia operating system imposes severe restrictions on the rate of thread activation.

At first glance, this may seem overly pessimistic. After all, even under Solaris, it takes almost six thousand context switches per second to consume a tenth of second of processor time and it's hard to believe that running a handful of multimedia applications needs thousands of threads.

However, the problem with threads in a multimedia operating system is that these six thousand context switches actually represent very few threads. A multimedia operating system must be capable of providing streams of evolving media to its user. These streams, while discretized by the nature of computing, must present their constituent sample streams sufficiently quickly so that they appear continuous to a human user.[3]

For rendering graphics or displaying video, 60Hz frame rates provide a convincing illusion of continuity. Future multimedia systems may require even

[1] SPARC, Solaris and the workstation or processor names listed in Table are the property of Sun Microsystems. All other trademarks mentioned are the property of their respective owners.

[2] For example, MicroUnity's media processor provides multiple register sets on chip and hardware-assisted threading.[Mic96] This is however a very scarce resource.

[3] Abadi and Lamport observe in [AL92] that providing the illusion of continuous time in a discrete environment is the fundamental problem underlying realtime computing.

higher sampling rates. For example, processing audio streams to provide correct positional cues may require adjustment of the audio stream at 8kHz rates in order to provide a consistent illusion of HRTF-filtered sound bound to head-movements.[Bla83]

Maintaining a 60Hz frame rate requires that the rendering threads in each multimedia application have a chance to run in a basic period of 17milliseconds. Less than a hundred threads will consume more than a tenth of second of processor time just in context switching!

Perhaps this still seems like far too many threads for supporting a handful of multimedia applications. However, providing a high-degree of multiprogramming in an interactive application, particularly a multimedia application, has significant user interface advantages. Trestle [Man92], NeWS [GRA89] and HotJava [Sun95] all demonstrate the usability advantages of extensively multiprogrammed systems. For example, we can easily envisage a single video conferencing application consuming thirty to forty threads. The authors' preferred web browser, Sun's HotJava, averages more than twenty separate live execution threads during normal usage. A game might use a thread to compute the behaviour of each sprite.

The skeptic will observe that applications don't need to be such large forests of threads. Programs such as the NeWS window system server quite successfully provided the illusion of multi-threading without any underlying operating system thread support[SK95]. Instead, programmers can carefully divide the application into small fragments, each of which voluntarily relinquishes the processor upon completion. The application's core contains code which dispatches these fragments successively to produce the illusion of multi-threading. In this fashion, a complex application may efficiently provide multiprogramming support while using only one operating system thread.

We take this concept to its logical limit, noting how that these small fragments are identical to our concept of performers. They're small fragments of code. They relinquish the processor voluntarily. They must be executed repeatedly. Why should we force each application to replicate the performer dispatch code when the operating system can do this for all applications? Instead, we can simplify applications and improve system performance by centralizing this dispatch loop in the operating system. Clearly, this is an efficient alternative to extensive threading for allocating processor resources in a multimedia operating system.

This returns us to the difficult problem of computing the total execution time needed by all the performers. An easy solution is to fix the set of performers that the multimedia operating system will execute in advance. This limitation prevents the operating system from extending its functionality at runtime but does mean that all system components can be scheduled at design time. Essentially, this solution turns the multimedia operating system into a traditional cyclic executive. In fact, it's easy to see how a multimedia operating system is a special case of traditional realtime control application. After all, just like a multimedia application, a flight controller must process in many incoming streams of information and produce many separate streams of output. That

these inputs come from gyroscopes and pitot tubes and outputs go to aileron and rudder actuators is only a change of medium. The resource allocation issues remain the same.

While possibly appropriate for dedicated multimedia coprocessors such as the NV1[NVi95], statically fixing the functionality of the system takes away one of the most valuable features of a multimedia operating system. Being able to add performers to the system at runtime is essential to resolving another of the most difficult resource allocation problems facing a multimedia operating system: controlling network bandwidth and latency.

X-terminals clearly demonstrate a method of implementing distributed applications which, despite having having no support for dynamically extending the operating system local to the user, remain quite successful. Despite this success, the static nature of the X11 wire protocol makes intolerant of network latency.

Should we desire a distributed application which can tolerate a wider range of network latencies than demonstrated by X11 applications, the operating system for the user-local station must be able to run application code fragments. Soon, the Internet may use satellite linkages resulting in a minimum latency of several seconds. X11 becomes extremely unwieldy in these circumstances. Instead, an application must be able to push functionality into either the client or server in a similar fashion to the down-loadable content popularized by Java[Sun95].

Permitting applications to deposit fragments of their functionality into a multimedia operating system executing local to the user has another advantage besides mitigating the effects of network latency. It permits adaptation. A fragment of down-loaded code can easily measure its own execution time and adapt its needs to reflect the computation power of its host. Across a static protocol, where asynchrony is necessary to provide even a poor attempt at limiting latency, feedback control is much more difficult[AN90]. With the trend towards ubiquitous mobile computing, a multimedia operating system should provide a clean mechanism for performers to adjust their computational needs.

Since a modern operating system should be capable of concurrently executing multiple applications, it must have a technique of sharing the processor amongst each application's down-loaded executable fragments. The traditional sharing technique provides each application with a separate thread of execution. However, above we already argued how threads are either too costly to context switch or used sparingly as each application divides itself internally into performers.

Hence we return to the fundamental problem of resource allocation in an operating system providing performers: deducing in advance the execution time required for invoking an arbitrary set of performers.

We are implementing an operating system called LiquiMedia that attempts to resolve this problem with a technique we call *statistical scheduling*. Clearly, an operating system can easily determine the total processor time needed to execute an arbitrary group of performers if it knows how long each one of them takes to execute. Unfortunately, no automatic mechanism can determine each performer's execution time in advance. However, an operating system can *mea-*

sure the actual duration of a performer every time it's executed. Over many invocations, once for each basic period, the operating system can compute the mean and variance for a performer's execution duration.

Provided that future invocations of a performer exhibit similar execution behaviour as previous invocations, these statistics let the operating system compute the probability of some set of performers running to completion in a known time interval. Making this assumption has enormous ramifications on the design of LiquiMedia. Clearly, not all algorithms or applications are appropriately implemented as performers. However, we believe that the interactive portions of multimedia applications are, in fact, ideally implemented in this fashion. In the interests of system reliability, we are currently investigating heuristic techniques to evaluate if code is likely to exhibit similar execution duration between invocations.

LiquiMedia does not need to verify a performer schedule as part of the performer dispatch cycle. Instead, it performs this computation and other maintenance activities during any free time remaining between the end of invoking all the performers and the end of each basic period.

Statistical scheduling operates in the following fashion. An application wishing to have the operating system invoke new performers requests that performers, be added to the schedule. Using stored statistical profiles of the performers, LiquiMedia computes the probability that a schedule including them will complete within one basic period. If the computed probability is less than a user-specified tolerance, the new performers are rejected and the application may request an operating-system mediated negotiation for a redistribution of computational resources.

Since any schedule verified in this fashion retains a finite probability of going overtime, LiquiMedia must successfully address the issue of failure detection and recovery. LiquiMedia's central dispatch loop handles the basic period's timer interrupt. The dispatcher can detect a schedule failure by examining its stack for evidence that it has been re-entered. If it has, it must attempt to recover. We are currently exploring different recovery strategies.

In our LiquiMedia prototype, performers have provided a useful alternative to threads for the allocation of processor time because discarding preemption obviates expensive context switches. Freed of this overhead, typically tens of microseconds per context switch on modern hardware, test performers in LiquiMedia have shown the high rates of function dispatch needed in a multimedia operating system. Furthermore, discarding preemption offers both a simpler architecture and considerable efficiency gains. Unfortunately, using performers to allocate computational resources exchanges the run-time overhead of threads for the problem of forecasting the run-time of an arbitrary set of performers.

Preliminary results show that statistical scheduling offers a useful mechanism for resolving this dilemma. We are developing a number of multimedia applications and expect the constituent performers to exhibit similar execution profiles across invocations. So far, coding performers in this fashion been an easy price to pay for the efficiency of a cyclic-executive style of operating system and freedom from expensive context switches. Consequently, we expect to

demonstrate that statistically scheduled performers are a useful way of providing fine-grain allocation of processor time in a multimedia operating system.

References

[AL92] Martin Abadi and Leslie Lamport. An old-fashioned recipe for real time. Technical Report 91, Digital SRC, Palo Alto, California, 1992.

[AN90] Susan Angebranndt and Todd D. Newman. The sample X11 server architecture. *The Digital Technical Journal*, 2(3), 1990.

[Bla83] J. Blauert. *Spatial Hearing: The Psychophysics of Human Sound Localization.* MIT Press, Cambridge, 1983.

[GRA89] James Gosling, David S.H. Rosenthal, and Michelle Arden. *The NeWS Book, An Introduction to the Network/extensible Window System.* Springer-Verlag, New York City, 1989.

[Man92] Mark Manasse. Trestle tutorial. Technical Report 69, Digital, Systems Research Centre, Palo Alto, California, 1992.

[Mic96] MicroUnity Inc. MicroUnity home page. http://www.microunity.com/, November 19 1996.

[NVi95] NVidia, Sunnyvale, California. *Product Overview: NV1*, 1995.

[SK95] Josh Siegel and Robert Kroeger. RBuss: a public domain NeWS clone. http://tenebre.uwaterloo.ca/NeWS/rbuss.html, October 1995.

[Sun95] Sun Microsystems Inc. HotJava home page. http://java.sun.com/documentation.html, September 19 1995.