ETI Resource Distributor: Guaranteed Resource Allocation and Scheduling in Multimedia Systems

Miche Baker-Harvey

Equator Technologies, Inc.
520 Pike Street, Suite 900
Seattle WA 98101-4001
miche@equator.com

Abstract

Multimedia processors offer a programmable, cost-effective way to provide multimedia functionality in environments previously serviced by fixed-function hardware and digital signal processors. Achieving acceptable performance requires that the multimedia processor's software emulate hardware devices.

There are stringent requirements on the operating system scheduler of a multimedia processor. First, when a user starts a task believing it to be backed by hardware, the system cannot terminate that task. The task must continue to run as if the hardware were present. Second, optimizing the Quality of Service (QOS) requires that tasks use all available system resources. Third, QOS decisions must be made globally, and in the interests of the user, if system overload occurs. No previously existing scheduler meets all these requirements.

The Equator Technologies, Inc. (ETI) Resource Distributor guarantees scheduling for admitted tasks: the delivery of resources is not interrupted even if the system is overloaded. The Scheduler delivers resources to applications in units known to be useful for achieving a specific level of service quality. This promotes better utilization of system resources and a higher perceived QOS. When QOS degradations are required, the Resource Distributor never makes inadvertent or implicit policy decisions: policy must be explicitly specified by the user.

While providing superior services for periodic real-time applications, the Resource Distributor also guarantees liveness for applications that are not real-time. Support for real-time applications that do not require continuous resource use is integrated: it neither interferes with the scheduling guarantees of other applications nor ties up resources that could be used by other applications.

1 Introduction

We have designed and implemented a multimedia resource manager and scheduler for managing resources on our MAP1000 processor. The processor is designed to execute applications that emulate such fixed-function hardware as MPEG video encoders and decoders, 2D and 3D graphics engines, audio devices, and modems.

The MAP1000 is a multimedia processor comprised of a Very Long Instruction Word (VLIW) processor with a RISC-like instruction set; a multi-element Fixed Function Unit (FFU); and a programmable, multi-ported DMA engine, called the Data Streamer. The VLIW processor can issue up to four operations each cycle. Figure 1 shows a block diagram of the MAP1000. [1]

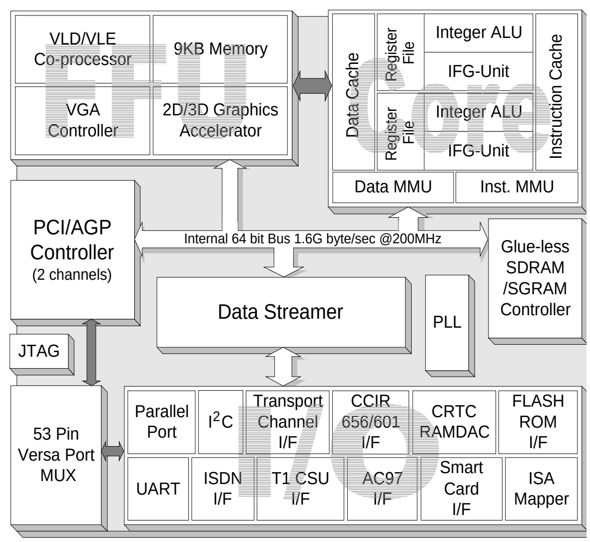

Figure 1: Block Diagram of the Map1000.

The MAP1000 runs the MMLite operating system, part of the Talisman program's Escalante reference design from Microsoft. The operating system supports basic functions with a COM-style interface. As delivered, the operating system supported a subset of the features based on the Rialto scheduler. We have

[1] Several additional features make the MAP1000 extremely effective at running multimedia applications. However, since these features do not raise issues for the Resource Distributor, they are not discussed in this paper. For more information, see [Basoglu et al 99].

To appear in the Proc. of the Third Symposium on Operating Systems Design and Implementation, February, 1999.

removed these features and implemented the ETI Resource Distributor (RD) in their place.[2]

Our goal is to provide an environment in which applications can emulate hardware. We support real-time guarantees that are harder than conventional soft real-time guarantees: user expectations for hardware do not allow for failures nearly as often as would occur with a truly soft real-time scheduler. We support conventional tasks running concurrently.

We provide absolute scheduling guarantees in the face of a dynamic task set. A real-time task admitted to our system is guaranteed to receive predictable resources. This guarantee applies in every period, whether the system is overloaded or not.

Decision making about resource distribution policy is well-integrated. Policy decisions are made globally, not by any single application. The policy delivered is affected neither by accidents of timing nor by the order of task creation. Scheduling guarantees are maintained while policy decisions are being made.

While the RD was conceived in an environment that required harder real-time guarantees, it is appropriate for many systems that combine conventional and soft real-time tasks. Its stricter real-time guarantees, and its integrated support for making global policy decisions, make it attractive for real-time applications. Conventional tasks exist concurrently, and their performance is also constrained by the global policy.

2 Organization of This Paper

Section 3 of this paper describes the design of the ETI Resource Distributor (RD). Section 4 describes implementation algorithms for the primary RD components. Additional features and ancillary issues pertaining to the ETI RD are detailed in Section 5, and Section 6 quantifies performance. Section 7 draws conclusions and presents opportunities for future research and development in the area of multimedia resource managers and schedulers.

In this paper, the term *application* refers to some piece of code started by a *user*, where the *user* is someone sitting at the system. The application may be comprised of one or more *threads* or *tasks*, terms used interchangeably. Both are schedulable entities on the MAP1000.

3 ETI Resource Distributor Design

This section describes various aspects of the ETI RD design. First, we attempt to characterize multimedia applications that run on the MAP1000. We then itemize design requirements abstraced from these applications. Next, we present ETI RD design components and follow them with a discussion of alternative designs. We conclude the section with a synopsis of the benefits offered by our design.

3.1 Application Characteristics

The ETI RD manages resources in a way that maintains for the user the appearance of actual hardware rather than of software running on a separate processor. In particular, applications must degrade gracefully in overload, and one application cannot cause unpredictable behavior in another. MAP1000 applications share some of the following characteristics that helped to drive this design:

1. They are primarily periodic, with naturally or externally set periods.

Some applications are strictly periodic, such as MPEG, whose period is defined by the content provider. Others have a period defined by actions that occur only at specific points; for example, the output for 2D graphics is paced by the screen refresh rate set by the user. While the periods we support may be as small as 500 µSec, the CPU requirements within a period tend to be relatively high. For example, the AC3 audio task requires about 12% of the core VLIW processor cycles.

2. They are capable of shedding load when the system is overloaded.

The applications can shed load when there are insufficient resources to run all tasks at the highest quality. For instance, the MPEG decoder can drop frames or change resolution to use fewer resources.

3. Their resource requirements are discrete, not continuous.

The work that an application must do is predictable. For example, processing an MPEG frame to a given resolution requires a known amount of CPU time. If more is allotted, it will be wasted; if less is allotted, the frame will not be decoded in time. To shed load, the MPEG application drops some complete frames or alters the resolution of the output. It is not useful for MPEG to process part of a frame.

Resource requirements for the MPEG application take quantum steps, from a maximum CPU requirement for top quality (displaying all frames at full resolution) to a lowest CPU requirement for the poorest quality supported (dropping frames and/or reducing resolution). The application can make only step-wise degradations.

This is not the case for all applications. Both 2D and 3D graphics are notable exceptions: the work they must do is a function of the complexity of the scene to be rendered, which is not known far in advance.

4. Their performance is extremely well characterized.

At a low level, the performance characteristics of these applications are well understood. On the MAP1000, the inner loops are understood down to the cycle.

[2] Information on the MMLite operating system can be found in [Jones et al. 96] and [Helander & Forin 98].

The task set is dynamic, with most new applications being started by user request. Generally, these applications can be denied service if the system does not have the resources to run them. Some applications cannot be denied service, or the user does not want them to be denied service. A telephone-answering modem task is an example.

3.2 Design Requirements

The ETI Resource Distributor has three design requirements (or "first principles") derived from the nature of our multimedia applications and user requirements. First, the RD must not cause performance anomalies in tasks that were started by the user. Second, it must allocate (nearly) 100% of available resources to ready tasks. Third, the quality of service (QOS) policy is determined by the user. Each of these requirements is now described in further detail.

1. Once a task has been successfully started by the user, it must continue (from the user's perspective) until it terminates naturally or is terminated by the user.

Assume that the user initiates a task (e.g., hitting the "play" button on the CD player). Once the task has begun (e.g., the sound track begins to emanate from the speakers), it will continue until it terminates (the end of the CD is reached) or until the user terminates it (e.g., by hitting "stop"). The application should behave as a user expects dedicated hardware to behave.

2. The scheduler must allocate (nearly) 100% of available resources to ready tasks.

If a task is ready to run and some resource is partially unused, it will be made available to that task. For example, idle CPU time will be granted to a requesting task. If a task requests a resource that an earlier task reserved but is not using, the later task will be granted that resource if scheduling guarantees can still be met.

3. Quality of service modifications should be made in response to user requirements.

Assume that a system is overloaded because too much of one resource is required by the task set. A task must either be terminated (which is inconsistent with the first principles) or asked to shed load by providing a lower QOS. The decision as to which task(s) should be asked to shed load, and by how much they should be asked to degrade their service, should be based on user preferences. There must be a global understanding of how the task set is meeting the users' needs: QOS decisions should not be made (solely) in the context of a single thread.

3.3 Components and Control Flow of the Resource Distributor

The ETI Resource Distributor consists of three components: the Resource Manager, the Scheduler, and the Policy Box. Figure 2 shows these components and the communication paths among them.

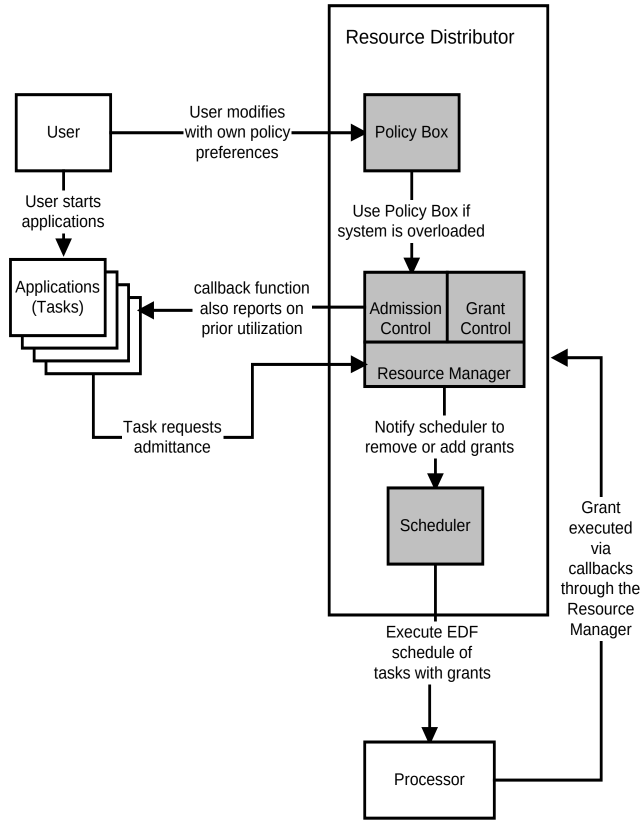

Figure 2: Components of the Resource Distributor.

The key insight that led to the design of the RD concerns the nature of application resource requirements: the QOS degradations that the applications can actually do are discrete. Performance levels are step-wise and known in advance by application writers. Resource allocations that do not map to a known service level in the application will result either in a missed deadline or in unused resources.

The *Resource Manager* allocates system resources to competing tasks. Allocation decisions are broken into two parts. The first, *admission control*, is the process of determining whether a thread can be admitted to the system. An admitted thread is guaranteed some non-zero amount of resources every period. Our approach requires that applications present a list of the load-shedding possibilities that they support at the time they request admittance. When the Resource Manager makes decisions, it thus has complete knowledge of the possibilities in the system.

The second part of the allocation decision, *grant control*, determines how much of each resource will be given to each thread (i.e., the *grant set*). The grant consists of a time period and an amount of resources that can be consumed in that time. For instance, a grant might allocate 10 ms of CPU cycles in a 30 ms period. The grant is a guarantee to the thread that this much resource will be allocated to the thread in each period.

The work of actually ordering the threads to be run, and guaranteeing that granted resource allocations are delivered, is performed by the *Scheduler*. The scheduler makes no policy decisions: it simply cycles through the set of threads that has been established by the grant control process.

The Scheduler implements an Earliest Deadline First (EDF) schedule [Liu & Layland 73] of the tasks that have been specified by the Resource Manager. It enforces the grants that have been made by the Resource Manager, limiting applications to their resource allocation if there are other applications that are also ready to run. The Scheduler accepts grant modification information from the Resource Manager and implements the changes in a way that maintains the scheduling guarantees of the first principles; it also passes accounting information to the Resource Manager. The Scheduler communicates only with the Resource Manager -- never with the Policy Box, with users, or with any application.

When the Resource Manager is unable to give each application all the resources it has requested, it refers to the *Policy Box* to resolve the conflict. The Policy Box is a repository of information on how to make tradeoffs among the QOS possibilities for the different applications that are running. The Policy Box contains default settings, but these settings can be overridden by users.

3.4 Alternative Multimedia Schedulers

Existing multimedia schedulers address some of the requirements posed by a multimedia environment. However, none of them meets all the requirements or addresses all the first principles.

Other multimedia schedulers support both soft real-time and interactive tasks. Most are designed for use in a PC or workstation environment, where one or more users are doing conventional workstation tasks and there is multimedia activity. This differs from our environment in that we must supply tighter real-time guarantees to our multimedia tasks, and some of the interactive requirements of the workstation environment are absent.

All multimedia schedulers also acknowledge the need for QOS reductions in overload. When there are more demands for resources than can be met, the schedulers expect the applications to shed their load and recover gracefully.

Processor Capacity Reserves, from CMU [Mercer et al. 94], provides CPU reservations on a per-thread basis. Reservations are enforced, so a poorly behaved task cannot impinge on a well-behaved task with a reservation. Reservations can be passed between tasks in a multi-threaded environment. The actual scheduling algorithm used is Earliest Deadline First (EDF) and is based on the period for which a reservation is made.

The SMART Scheduler from Stanford [Nieh & Lam 97, Nieh & Lam 96] provides for the simultaneous execution of conventional and real-time tasks. It uses a modified best-effort scheduler. In underload, SMART meets all real-time constraints. In overload, conventional tasks continue to make progress, but real-time requirements are not necessarily met. Interactive tasks get good response, but low-priority threads may be denied service.

The Rialto system was developed at Microsoft Research [Jones et al. 97, Jones et al. 96, Jones et al. 95]. It combines resource reservation and constraint-based scheduling in an aggressive system that tries both to manage overload gracefully and to handle networked, independently authored, real-time environments. To better support cooperating processes, the scheduling algorithm uses minimum-laxity scheduling, with a concept of virtual time. The intent is to let cooperating, independently authored tasks reason about their real-time requirements.

None of these approaches meets all of the ETI RD's design requirements. Processor Capacity Reserves encourages applications to over-reserve, so the full processor may not be used. Both SMART and Rialto have approaches to avoid this, but they do not handle overload as required by our first principles. In SMART, overload is handled with fair-share scheduling, which conflicts with the discrete resource requirements of our applications. In Rialto, the nature of constraints causes the system to make policy decisions after a deadline may have already been missed. We discuss these implications in more detail later in the paper.

3.5 ETI Resource Distributor Benefits

The ETI RD is a real-time scheduler and resource manager that provides stronger guarantees than those provided by other soft real-time schedulers. It strictly adheres to the three first principles. As a result, it provides:

1. Better admissions control
2. Better resource allocation
3. Clear, practical separation of scheduling and QOS decision-making

Better Admissions Control

The RD guarantees its admissions: An admitted task is guaranteed that it will not miss a deadline. Most other systems cannot make this guarantee because of transient overload conditions. Examples include SMART, which does fair-share scheduling in overload, and Rialto, which does not guarantee that a repeating constraint will be met in advance.

The RD does not reserve resources for a task that is not actually using them. Some systems, such as CMU's Processor Capacity Reserves, can provide guaranteed admission; however, these systems foster the over-reservation of resources so that deadlines can be met.

The RD also provides support for tasks that are not currently using resources but which cannot be denied admittance at some unspecified later time. These tasks, called *quiescent tasks*, are supported while still providing guaranteed admissions for non-quiescent tasks.

Better Resource Allocation

Unlike resource reservation schemes, the RD allocates 100% of available resources for ready tasks. The allocations are specifically tailored to the needs of the applications that are running. Unlike fair-share or best-effort schedulers, the RD allocates units of resources known to be useful to a thread. Resource allocations are made in quantum units; hence, resources are not allocated to a task that cannot meet its deadline. The actual resource allocations made are determined by the user. No accident of timing plays a part in the QOS provided by an application.

Clear, Practical Separation of Scheduling and QOS Decision-Making

The RD provides effective, reliable separation of scheduling and QOS decision-making. QOS decisions are always made in a global context, with reference to a user-defined Policy Box. The RD makes QOS decisions only during the time that has been allocated to the task requesting a global change in resource allocation. The RD's policy decisions are never made either in the context of a single application or when a thread will miss a deadline.

Alternative soft real-time systems deny users control of QOS decisions. There is usually some attempt to separate resource management, scheduling, and QOS decision making, but it does not go far enough. A system that makes any of its scheduling or resource allocation decisions in real-time fails in this regard when the system is in overload. Some of the literature acknowledges this clearly [Nieh & Lam 97], while other literature has not dealt explicitly with the issue of how to perform resource allocation and scheduling in overload conditions.

Most systems provide a failure notification to the application that is requesting resources. This independently authored application may then shed load. The problem with this approach is that the application that has just been denied service was selected by an accident of timing. The user might instead prefer that some other application degrade its service.

Alternative systems could assume a set of well-behaved applications, which would refer to a global third party to determine who should shed load. However, three problems remain. First, by the time the response returns from the third party, the deadline may no longer be reachable. Second, there is nothing in the literature that addresses how some other selected task, or the scheduler, would be informed that it is required to degrade its service. Third, even if another thread could be notified, it might either fail in the current frame or not degrade its service until later, causing other threads to miss their next deadlines.

4 RD Implementation

We now present the algorithms used by the Resource Manager, the Scheduler, and the Policy Box. We address some of the key issues in each area.

4.1 Resource Manager Algorithms

An application seeking real-time guarantees must access the Resource Manager to "request admittance." The application passes a resource list to the Resource Manager. The resource list is an ordered list of entries, each of which corresponds to one level of QOS that the application can provide.

Each entry contains a period and CPU requirement, both of which are specified in units of 27 MHz ticks. The reason for using the 27 MHz tick as a unit relates to clock synchronization issues and the MPEG tasks, as explained later in the paper. The minimum period is 500 μSec, and the maximum is 159 seconds. Each entry also contains a callback function associated with that level of QOS.

Table 1 shows the simplified format of a resource list. It omits several fields that manage resources other than CPU cycles on the MAP1000.

The "period(s)" selected by the applications are either naturally occurring, defined by some standard, or set by external events. For instance, MPEG needs to generate 30 frames per second, so a period of 1/30th of a second is needed. Expressed in terms of 27 MHz ticks, MPEG requests a period of 900,000 in its resource list. If the user in a PC environment had selected a display refresh rate of 72 Hz, a period of 375,000 ticks (27,000,000/72) would be used by 2D graphics. The period defines the start and end times of each time unit in which resources are allocated for this application. The start of period (n+1) is the same as the end of period (n).

The "CPU requirement" is a measure of the amount of CPU time the application requires during every period. If MPEG requires 1/3 of the CPU, it would pick a CPU requirement of 300,000 ticks. The "Rate" value is computed as the CPU requirement/ period and represents the rate at which the resource list entry consumes CPU resources.

Period 27 MHz	CPU Req. 27 MHz	Rate (computed)	Function
Period_{max}	CPU Req. $_{max}$	CPU Req. $_{max}$ /Period	Func_{max}
Period_j	CPU Req. $_j$	…	Func_j
Period_i	CPU Req. $_i$	…	Func_i
…	…	…	…
Period_{min}	CPU Req. $_{min}$	CPU Req. $_{min}$ /Period	Func_{min}

Table 1: Simplified Format of a Resource List.

"Function" is the address of a routine that implements the level of QOS appropriate for the resource list entry. An application may have different functions for different entries. The scheduler upcalls to the function when the application has been granted the resources associated with the resource list entry.

Example resource lists for MPEG decoding and 3D graphics are shown in Tables 2 and 3, respectively. The MPEG application sheds load by selectively dropping more B frames. The 3D graphics application sheds load simply by making less progress on the same function. Neither of these examples necessarily reflects how real applications would shed load. The resource lists again are simplified.

Period	CPU Req.	Rate	Function
900,000	300,000	33.3 %	FullDecompress()
3,600,000	900,000	25.0 %	Drop_B_in_4()
2,700,000	600,000	22.2 %	Drop_B_in_3()
3,600,000	600,000	16.7 %	Drop_2B_in_4()

Table 2: Resource List for an MPEG Thread.

The resource list for a thread does not change if it is quiescent. A quiescent thread is not scheduled, because it is in a different mode. It is not necessary to add a null entry to the resource list for a quiescent thread.

When a task requests admittance, the Resource Manager performs two distinct tasks. The first, admission control, determines whether the application is allowed to run with scheduling guarantees: to be admitted to the domain of the ETI Resource Distributor. If the task is admitted, the Resource Manager performs a second task, determining the new grant set. The grant set determines which resources are made available to each admitted task.

Period	CPU Req.	Rate	Function
2,700,000	2,160,000	80%	Render3DFrame()
2,700,000	1,080,000	40%	Render3DFrame()
2,700,000	540,000	20%	Render3DFrame()
2,700,000	270,000	10%	Render3DFrame()

Table 3: Resource List for a 3D Graphics Thread.

A new thread is allowed to enter the system if and only if the sum of the minimal grants for all threads (runnable and quiescent) in the system can be simultaneously accommodated if the new thread is admitted. The admissions control test is expressed as:

$$\sum_{i=0}^{Runnable} Rate(\min)_i + \sum_{j=0}^{Quiescent} Rate(\min)_j \leq 100\%$$

When a thread enters or leaves the system, or when a potentially quiescent thread changes state, the Resource Manager generates a new set of grants for all threads. The grant for a thread can increase or decrease at this time. The thread is informed indirectly, because the next period is started with a call to the function associated with the new grant. Because the Resource Manager ensures that the sum of grants does not exceed 100%, the scheduler need only enforce the grants to be able to use a simple EDF scheme to successfully schedule all threads. The calculation used to determine the new grant set is:

$$\sum_{i=0}^{Runnable} Rate(grant)_i \leq 100\%$$

Because we admit a task only if the sum of the minimum resource list entries is less than the total resources available on the machine, we are assured that a legitimate grant set exists: at worst, all tasks receive their minimum grant. If possible, all tasks are given their maximum grant. However, if there are insufficient resources, the Policy Box is referenced to make trade-offs. An example grant set for three tasks is shown in Table 4.

	Period	CPU Req	Rate	Function
Modem	270,000	27,000	10%	Modem
3D	275,300	143,156	52%	Render 3DFrame
MPEG	810,000	270,000	33%	Full Decompress

Table 4: Grant Set for Three Threads: Modem, MPEG Decompression, and 3D Graphics.

The Resource Manager does its work in the context of the requesting application. Admission control is performed only when a new task tries to enter the system.

A new grant set is computed only when a task enters or leaves the system, when it changes its resource list, or when it enters or leaves the quiescent state.

One strength of the Resource Distributor is that the cost of computing a grant set is never paid using cycles that have already been committed to some other admitted task. By design, the costs of the Resource Manager cannot affect its ability to meet its scheduling guarantees. The Resource Manager decisions are made neither in interrupt mode nor when a deadline is in jeopardy. Coordinated communication of the new grant set to the Scheduler is done so that scheduling guarantees are maintained.

4.2 Scheduler Algorithms and Guarantees

The Scheduler implements an Earliest Deadline First (EDF) scheduling algorithm. The EDF algorithm is proven to be able to schedule any set of tasks for which there is a schedule. By not allocating more resources than there are, a set of tasks is guaranteed to be schedulable. EDF is extremely cheap to implement [Liu & Layland 73].

There are some rules governing the ability of an EDF scheduler to make scheduling guarantees.

1. *All tasks must be periodic*.

All tasks managed by the Resource Distributor are periodic tasks. Sporadic tasks are managed by a Sporadic Server, as discussed below.

2. *All tasks must be infinitely preemptible*.

While tasks are not infinitely preemptible, they are finely preemptible. In fact, to minimize context-switch overhead, we override the EDF policy when the overlap between two tasks is extremely small. If the currently executing thread has a distant deadline but only a small allocation of CPU time remaining, we complete it, even though another thread with a nearer deadline is runnable. The length of this override time is a function of the context-switch time.

3. *The period end is the period start*.

EDF does not support a separate start time and period start. This implies that an application must be willing to take its allocation at any point within its period.

4. *There can be no synchronization between tasks*.

The reason for this limitation is the same as the previous: a task must be willing to accept its allocation at any point in the period. If a task has blocked to synchronize, it might miss the (only) window in which it could be scheduled. Non-blocking synchronization is acceptable.

One implication of EDF is that the maximum guaranteed latency for a task is twice its period minus twice its CPU requirement. This occurs when the grant is delivered to an application at the beginning of one period and at the end of the subsequent period.

The Resource Manager notifies the Scheduler that a new grant is available. The next time there is unallocated CPU time, the Scheduler makes a callback to the Resource Manager to get the new grant information. By waiting for unallocated time to begin a new grant, we assure that adding a new task cannot affect the running of an already admitted task. The Scheduler is notified immediately that a grant should be removed or decreased, and the decrease occurs in the next period for the affected task.

The Scheduler maintains all tasks with grants on one of two queues: (1) The TimeRemaining queue, which contains all tasks that have unused CPU cycles allocated in this period, or (2) the TimeExpired queue, which contains all others. Both queues are ordered by deadline. A thread on the TimeExpired queue has either used its allocated CPU cycles for the period or indicated that it is done with its work for the period. A thread on the TimeExpired queue can also be on an Overtime-Requested queue if it ran out of time and still had more work to do.

On a context switch, the Scheduler takes the first thread off the TimeRemaining queue, if there is one. If no threads have time remaining but there are new grants, it calls back to the Resource Manager to get the new grant information. Finally, it takes the first thread off the OvertimeRequested queue. We always maintain at least one thread (the Idle thread) on this list.

The Scheduler sets a timer interrupt for the next context switch. This occurs at the earlier of: (1) the end of the grant for this thread for this period, or (2) the beginning of a new period for another thread whose next-period end precedes the period end for the thread about to run.

The next context switch is caused either by a timer interrupt or by the running task yielding the processor.

The ETI RD takes exactly those context switch interrupts required by the set of applications running on the system. We take (at least) twice as many interrupts as the shortest period in the system. If a task has a period of 5 ms, we switch context at least twice every 5 ms. The number of context switches can be minimized when tasks have the same period or periods that are multiples of each other, but this is an artificial restriction for most task sets.

Figure 3 shows a schedule for the grant set depicted in the example in Table 4. The EDF schedule preempts the MEPG and 3D Graphics tasks.

The ETI Resource Distributor makes the following scheduling guarantees to admitted tasks:

1. The task will receive a grant from the Resource List supplied by the application.
2. The grant will be delivered in each period.
3. Unless the task has the smallest CPU requirement in the system, it may be preempted each period.
4. The grant will not change mid-period.

5. The task will not be involuntarily terminated.

These guarantees are void for any period in which the task is blocked, but they will resume in the first full period in which the thread is not blocked.

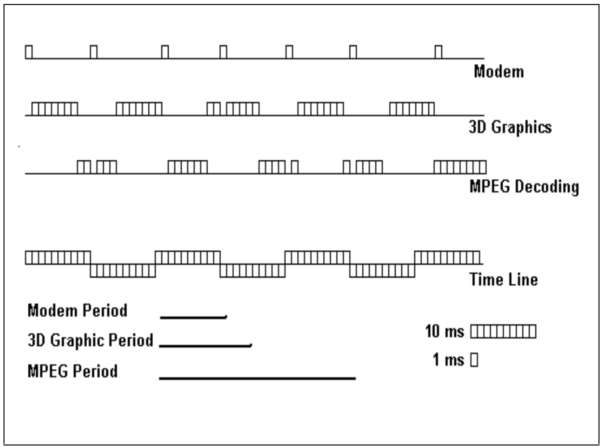

Figure 3: Schedule for Modem, 3D Graphics, and MPEG Decode.

4.3 Policy Box Algorithms

The Policy Box is a repository of information used to resolve conflicts when the system is overloaded. It is accessed by the Resource Manager when: (1) system requirements change, and (2) not all tasks can have their maximum resource list entry.

Because the Policy Box is always consulted when the system goes into overload, policy decisions are never made locally. Further, because decisions are made in the context of a task not yet admitted, they are not made when a deadline is about to be missed. The policy is not even affected by the order in which a set of threads is started. The cost of consulting the Policy Box is paid by the task requesting admittance. Therefore, no task will be denied service or miss a deadline as a result of the system going into overload.

The Policy Box has default policies supplied by the system designers, which can be overridden by users. For example, video should generally be degraded before audio, because most users are more sensitive to the quality of audio. However, in a loud environment, the clicks and pops of poor audio may be indistinguishable from ambient noise. In such an environment, where the user may rely more on video than audio, the quality requirements may be reversed. Alarms (with clicks and pops) are still needed, but the visuals must be current.

Each policy contains a relative ranking for its included threads. These rankings are used to compute which resource list entry each thread should receive. A simplified example Policy Box is shown in Table 5. In this example, there are four tasks known to the Policy Box. The Policy Box correlates a task name and Policy Box identifiers.

Policy ID	**Rankings**			
	Task 1	**Task 2**	**Task 3**	**Task 4**
1,2	10	85		
1,3	20		75	
1,4	10			85
1,2,3	10	50	35	
1,2,4	10	35		50
1,3,4	10		35	50
1,2,3,4	5	35	20	35

Table 5: Example Policy Box.

5 Additional Features

This section discusses important additional features and characteristics of the system. We describe our support for other task types, timing issues, the specifics of grant delivery, and preemptions.

5.1 Sporadic Tasks

Most tasks in our system are periodic and have real-time characteristics. Some tasks are neither periodic nor real-time. We call these *sporadic tasks*.

Sporadic tasks are managed by a Sporadic Server, which is itself a periodic task [Sprunt et al 89]. We provide an interface whereby any periodic task can "assign" its grant for a specific period of time to another (non-periodic) task. The Sporadic Server maintains a round-robin queue of the sporadic tasks in the system. When it is scheduled by the Scheduler, it looks for work to do on its queue. If there are sporadic tasks ready to run, the Sporadic Server assigns its grant for some fixed amount of time (currently 10 ms) to them. The Sporadic Server then returns to the Scheduler.

When the Scheduler selects a periodic task to run, it routinely checks to see if the task's grant has been assigned. If it has, the Scheduler runs the assigned-to thread instead. Resource bookkeeping is still done in the context of the periodic task. The assignment extends over multiple periods if more time is assigned than is available in a single period. When the grant is consumed, or when the sporadic thread blocks, the Scheduler returns to the periodic task.

Sporadic tasks can perform the same functions as periodic tasks. The only distinction is that there are no scheduling guarantees for a sporadic task. The performance of a sporadic task is a function of the amount of CPU time allocated to the Sporadic Server (which can be modified through the Policy Box) and the number of sporadic tasks.

5.2 Interrupt Tasks

The periodic model does not work well for low-latency, high-frequency tasks. Latency requirements of less than about 1 ms cannot be accommodated by periodic tasks running under the ETI Resource Distributor. (Recall that the best guaranteed latency is two times the period minus two times the CPU cycle allocation.) Generally, these tasks are triggered by an interrupt and must be serviced by an interrupt handler.

Because these tasks do not come under the purview of the Resource Distributor, their resources are not taken into account. We reserve a small percentage of the processor for handling interrupts.[3] Tradeoffs must be made between keeping this number small to avoid wasted resources and making it large enough that interrupts do not conflict with the deadlines for admitted tasks.

5.3 Quiescent Tasks

A quiescent task is one that is not currently using any resources, but which cannot be denied admittance when ready to run. One example is a cool-down task. If the processor is overheating, the operating system is notified and is expected to cool down the processor. It does this by running a no-op loop that switches fewer transistors. The cool-down task does not need to use 100% of the processor (because if it did, it might make more sense to shut down the system); however, it does need some percentage, depending on the extent of overheating. Until the processor has overheated (if ever), we do not want to reserve resources for this task. However, if the processor has overheated, we must run the cool-down task.

An obvious, simple way to handle this situation is to terminate some other task that is running. However, this approach violates our scheduling guarantee.

A better way is to incorporate the quiescent task into the calculation for admissions control and to ignore it for calculating the grant set. With this approach, when the task ceases to be quiescent, we are guaranteed a grant set for all admitted tasks: at worst, all tasks receive their minimum resource list entry. However, while the task is quiescent, the resources it would otherwise use are allocated to other tasks, enabling them to provide a higher level of QOS.

[3] On the MAP1000, the Data Streamer greatly reduces the number of interrupts taken. Most tasks are double or triple buffered. In period (n+1), they may be outputting the data generated in period (n). This, combined with the fact that the Data Streamer can be programmed to do flow control, reduces the number of I/O interrupts in the non-error case to near zero.

An example is a telephone-answering modem task. Imagine a PC environment where the user is studying multimedia data from a DVD. The user is waiting for a teleconferencing connection to be established so that the multimedia data can be discussed. Until the telephone call occurs, the full resources of the machine should be dedicated to the DVD. Afterwards, the modem, teleconferencing, and DVD software must share resources, and the DVD may have to shed load. Our Resource Distributor lets the user start these applications in any order. The desired global policy decisions will be made, the correct load shedding performed, and the telephone answered promptly.

5.4 Clock Synchronization Issues

The periods of applications are frequently defined by external clocks. For instance, MPEG applications are tied to the TCI clock, which runs at 27 MHz [ISO 96]. The TCI clock is routinely modified by the MPEG system software to stay synchronized to an incoming MPEG data stream. In other words, it is known that the MPEG clock will drift with respect to any other clock. The system clock on the MAP1000 is 200 MHz.

Clocks driven by different crystals can drift with respect to each other. One can be running slightly faster, one slightly slower. The TCI clock can do both. Sometimes it drifts faster, sometimes slower, depending on the source of the MPEG input stream.

The drifting of two clocks with respect to each other causes problems for periodic threads whose period is externally defined. For example, imagine that a user has selected a 100 Hz display refresh rate. Every 10 ms the MAP1000 needs to have an image ready to display, and every 10 ms the Display Refresh Controller (DRC) picks it up. Since the image-generating application on the MAP1000 and the DRC are controlled by different clocks, they can drift with respect to each other. In time, one of them can get an entire frame ahead of or behind the other. At this point, either an entire frame is dropped, or a frame is displayed in duplicate.

Unlike the case for many applications, the DRC can ignore this problem at a fairly small cost. It is commonly the case that the same screen is displayed multiple times, since the MPEG frame rate is typically less than the refresh rate of a non-interlaced RGB computer monitor. Further, there is unlikely to be anything unique about a single display frame that would make its loss seriously reduce quality. A more significant problem is tearing: if the DRC displayed half of one frame and half of the next, the user could detect a quality degradation. This problem is common with most schedulers. It is usually avoided by changing the pointer to the data to be displayed only when it is complete. For the DRC, clock synchronization issues are relatively easy to manage.

However, some other applications have significant problems with clock synchronization. One example is an MPEG decoder. The MPEG data stream is received live, at 30 frames per second. The stream is compressed, and not all the delivered frames are the same type. An MPEG stream has I, B, and P frames. The I frames are Initial frames without temporal prediction: they can be decoded in isolation. The compressed P frames are encoded as the difference from the previous I or P frame. The compressed B frames are encoded as the difference relative to both the preceding and following I or P frames. The significance of losing a B frame is small – one frame is not displayed. However, if an I frame is lost, a perfect picture cannot be displayed again until the next I frame is received, which is typically every 15 frames or half-second. A half-second loss of video is noticeable and unacceptable. Therefore, for MPEG, if an I frame is lost, the QOS would be inadequate.

Because the consequences of losing an I frame are unacceptable, we have partially finessed the problem of staying synchronized with the TCI clock by using the TCI clock for scheduling. This means that the first MPEG transport stream does not need to worry about TCI synchronization. However, a second MPEG thread using a different TCI transport stream, and any other application that wants to stay synchronized with an external clock, must do so in software.

We provide an interface (InsertIdleCycles) that can be used to postpone the start of the next period for a task by an arbitrary number of 27 MHz ticks. Postponing the period cannot jeopardize the scheduling guarantees to other tasks, but pulling it in would, so the interface cannot be used to pull in the period start. This interface can be used both to control the effects of drift from clock skew and to get into phase with a clock. The application must read both the TCI and the external clock at some interval. The difference between the external clock readings is determined. From that, the expected difference in the TCI clock is computed. The actual difference in the TCI clock readings can be used to calculate the skew.

5.5 Semantics for Delivering a Grant

When a task receives a grant, we make a callback to the function named in the resource list entry. The stack is cleared before the call, and the calling arguments include whether the previous call completed, the sum of the resources used in the previous call, and an indicator of which grant has been assigned for this period. This is how the initial grant for an admitted task is always delivered.

We offer both callback and return semantics for all periods past the initial one. For truly periodic tasks (e.g., MPEG, modem, audio), a callback is generated at the beginning of every new period. The work of the previous period is complete, and the exact same function is performed on the set of data in this new period. For tasks such as 2D and 3D graphics, which are not as tightly tied to their periods, the state between periods should be retained, and the application should continue where it left off. These tasks use return semantics. Note that all tasks use return semantics when they have been preempted in the middle of their grant for the period; callback semantics apply only at the beginning of a new period.

A task must be prepared to receive a new grant in any period. For tasks using callback semantics, this is trivial. For those using return semantics, some clean-up operations may be required. Depending on the differences between the old and new grants, the task may even prefer to use return semantics on the new grant.

An example is the 3D graphics task, which has multiple resource list entries that use the same function. On the MAP1000, the 3D graphics application has some resource list entries that use the video scaler component of the FFU, and some that do not. If the grant change involves either acquiring or losing access to this unit, then the 3D graphics task needs to use callback semantics to get the grant started after some clean-up. If the access to the FFU does not change, it uses return semantics.

We provide a filter callback option for a task that uses return semantics when its grant changes. If the task has registered a filter callback, we call it instead of either returning or using the new grant's callback function. The task does whatever cleanup is necessary and then returns an indicator as to whether it would prefer return or callback semantics for this one call.

5.6 Preemptions and Real-time Applications

As long as the tasks in a system have differing periods and CPU requirements, it will be necessary to preempt the tasks with longer CPU requirements and periods. Preemptions are expensive and disruptive. Besides the context switch overhead, the cache state may also be lost.[4]

The majority of tasks we run are double or triple buffered. At the end of a buffer, some data will even be jettisoned. The best time to preempt a task is when it has finished handling one buffer and before it starts on the next. The "buffers" may be relatively small; they do not correspond to an entire frame, for instance, but perhaps to an MPEG macroblock, which is 384 bytes.

[4] Our chip has additional complications with the Data Streamer and the FFU. At any given point, both the Data Streamer and the FFU are likely to be performing long-running operations on behalf of the current task.

To minimize the cost of preemptions, we provide a mechanism with which a well-behaved task can perform *controlled preemptions*. Otherwise, it is preempted as usual.

We distinguish between involuntary and voluntary preemptions. An *involuntary preemption* occurs in a normal context switch, when the process is being given to another task. A *voluntary preemption* occurs when a task blocks, for example, on synchronization or I/O. A voluntary preemption also occurs when the task volunteers to yield the processor because the Scheduler must do a context switch.

To perform controlled preemptions, the task notifies the Resource Manager of its intention, and the Resource Manager notifies the Scheduler. When the Scheduler needs to preempt the task, it sets a marker indicating that a context switch is needed, notifies the task, and sets a timer interrupt for a grace period. Before the grace period expires, the task must notice that it is in the grace period and voluntarily preempt itself by yielding the processor. If the grace period expires before the task yields, it is involuntarily preempted.

The task can specify a local address at which it would like to be notified when a context switch is needed. By doing so, the task avoids the system call needed to determine if a preemption is required and may avoid additional cache misses. The grace period is quite short – on the order of a couple hundred μSec.

The grace period effectively lets one task run into the time allotted to another. The task will be charged for the resources it uses in the grace period; however, the other task is still postponed. Therefore, it is critical to keep the grace period as small as possible. On the other hand, the more often that applications have to check for preemption, the more constrained their coding, because they must be prepared to yield the processor. It remains a matter of further study to determine the optimal grace period length.

If a task is doing voluntary preemptions and fails to yield in the grace period, it is involuntary preempted. When next run, it is sent an exception callback, enabling it to clean up.

6 Performance

This section presents performance data for the costs involved in context switches, admissions, grant set determination, managing preemption and scheduling. Most costs are incurred in the context of a task either requesting admittance or significantly changing its state; the run-time costs are relatively small. All performance numbers reported in these sections were acquired on a cycle-accurate simulator.

6.1 Context-Switch Costs

One advantage of the ETI Resource Distributor is that preemptions are taken only when required for correctness of scheduling guarantees. The number of context switches depends on the number of applications, and the size of their periods and CPU requirements. Rialto reduces the number of context switches by enforcing the rule that all applications have periods that are even multiples of each other [Jones et al. 97]; we support any period length in range.

A context switch on the MAP1000 incurs the cost of saving and restoring as many as two banks of 64 32-bit registers. In our calling standard, most registers are caller-saved; therefore, for a synchronous (voluntary) context switch, only 14 32-bit registers (times two banks) must be saved. There are another 64 32-bit system registers that must be saved on an involuntary context switch.[5]

On a 200 MHz chip, a fully synchronous, voluntary context switch takes a minimum, median, and average of 11.5, 18.3, and 20.7 μSec. A fully involuntary context switch takes a minimum, median and average of 16.9, 28.2, and 35.0 μSec.

On a highly tuned system running an MPEG video decoder and AC3 audio, we might expect about 300 context switches per second (i.e., 60 each for the MPEG decoder and AC3 audio and for each of their data management threads, and another 30 for the Sporadic Server). Of these, 120 will be synchronous context switches, for a total of 2196 μSec. The remaining 180 context switches are asynchronous, at a mean cost of 28.2, for a total of 5076 μSec. For this load, we would expect a total context-switch cost of about 0.7% of the CPU.

Threads that have the same period do not preempt each other. If threads have different periods, the one with the shorter period can preempt the one with the longer.

6.2 Admissions Control Cost

Admissions control is computed in constant time. A running sum of the resources used for each thread's minimum resource list entry is maintained. When a new thread requests admittance, the resources of its minimum resource list entry are added to the running total and compared to what is available on the system. On a 200 MHz system, the admissions control process takes between 150 and 200 μSec.

[5] The cost of saving Data Streamer and FFU state is not considered part of the context-switch cost. For a well-behaved application, the cost is zero.

6.3 Determining a Grant Set Cost

The cost of determining a grant set is a function of: (1) whether the system is in overload, and (2) the number of threads admitted to the system. If the system is not in overload, we first make an O(1) determination as to whether every thread can have the resources requested in its maximum resource list entry. If so, we are done.

If the system is in overload, the computation becomes more complex. When the Resource Manager finds that not all threads can have their maximum, it asks the Policy Box for a policy for the set of admitted, non-quiescent threads. The Policy Box searches its database for a matching policy. If it does not find one, the current implementation invents a policy in which each of N threads receives 1/Nth of the resources, and an arbitrary thread is given control of exclusive resources.

Once it has received a policy, the Resource Manager correlates the policy received with the actual resource list entries of the threads. Our current implementation for this step is O(N). We iterate through each thread, noting the resource list entries just above and below the QOS specified by the policy. If the sum of the entries that were above the policy-specified QOS ratings fits, we are done. Otherwise, we walk through once more, turning higher entries into lower entries. This process will converge in a single pass, because only policies that fit are allowed by the Policy Box. We make a third pass if substantial resources remain unused after the second pass, looking for a thread that can use these otherwise unallocated resources.

6.4 Managing Preemption Cost

Threads that want to do controlled, voluntary preemptions make a call to the Resource Manager, giving a local address in which the notification should be placed. The cost of a managed preemption is potentially much less than the cost of an involuntary context switch. The application writer controls what information is in the caches and the states of the FFU and Data Streamer.

There are two incremental costs to doing controlled preemptions. First, the thread must periodically check to see if it is in a grace period. It must do this frequently enough so that it can get into a safe state and yield the processor before the grace period expires. On the MAP1000, the process of checking to see if a thread is in the grace period is essentially free. On any architecture with a functional unit that is not 100% utilized, an otherwise idle cycle on that unit can be used to check if preemption is required.

The second, and higher, incremental cost of controlled preemptions is incurred in the operating system, where we must notify the thread that it is in a grace period. When the operating system receives a timer interrupt, we check to see if the running thread is eligible for a grace period. If so, we set the location specified by the thread, reset the timer interrupt for the grace period, and continue the interrupted thread.

6.5 Scheduler Effectiveness

The ETI Resource Distributor effectively schedules a set of threads. We are currently running simultaneous sets of MPEG, AC3, and 3D graphics. There are a number of ancillary support threads for data management, for the Display Refresh Controller, etc.

The following data are from a test run with four periodic threads in addition to the Sporadic Server. Each is running with a period of $1/30^{th}$ of a second, and each has a maximum CPU requirements of 13, 2, 3 and 3 ms, respectively. The thread with the largest requirement never reports that it has finished its work for the period. This set of threads does not overload the system.

Figure 4 shows the schedule one-third of a second into the run. The two data control threads (8 and 10) are waiting for more data from the producers. Producer thread 7 receives the unused time (shown in the lighter lines) but is preempted when a new period begins; it then receives its guaranteed allocation (shown in the darker lines). The other producer thread (9) completes its work each period.

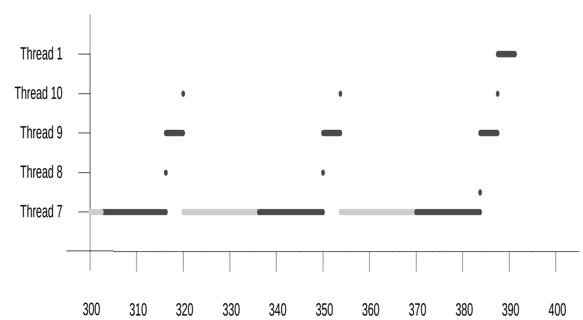

Figure 4: Schedule for Five Threads from 300-400 Ms. (Dark Lines show guaranteed allocations; light lines show allocations of unreserved time.)

This example shows a bug in the application: the data management threads should block, waiting for the data to become available. The context switches to the data management threads could be avoided when no data is available. The producer threads could set an event when data is available, and the data management threads would regain their scheduling guarantees in the following period.

The next test shows five threads in addition to the Sporadic Server, each of which has nine entries in its resource list. Each resource list entry has a period of 10 ms, and the nine entries range from requiring 90% to 10% of the CPU. Because there are no policies for these threads, we expect the Policy Box to make up a policy that evenly divides the resources among the available threads, including the Sporadic Server. The Sporadic Server requires only 1% every 100 ms, but it is the only thread that indicates it has work to do at the end of each period; the other threads all yield when preemption is required.

Each of the threads is started in turn, with a wait of 20 ms between each thread start. Because the period for each thread (except the Sporadic Server) is 10 ms, we expect each thread to be scheduled every 10 ms. As each new thread is admitted, we expect the CPU allocation for all previously admitted threads to be reduced. Since we reserve 4% of the processor for interrupt processing, and there are no interrupts other than those for the timer in this run, we expect the Sporadic Server to run at least every 10 ms. We also expect that each new thread will receive its first grant in a period that would otherwise have been given to the Sporadic Server as unallocated time.

Table 6 shows the resource list for each of the threads 2-6. Figure 5 shows that the run has behaved as expected. Thread 2 (the first periodic thread admitted after the Sporadic Server) begins with an allocation of 9 ms out of 10. It then drops to 4 ms when one thread is added, to 3 ms when there are three threads, and to 2 ms when there are four or five threads running in addition to the Sporadic Server. Each thread's allocations are received 10 ms apart, because the period does not change.

7 Conclusions and Future Directions

The ETI Resource Distributor is unique in its ability to provide scheduling guarantees to a dynamic set of applications without wasting resources. Like other soft real-time systems, we support the dynamic creation of tasks and the possibility of overload. Like hard real-time systems, we provide guarantees regarding the resources that will be made available to a task and the timeframe in which they will be provided. By designing for the step-wise nature of resource usage in multimedia applications, we support load shedding in a way that meets the user's performance requirements.

Period	CPU Req.	Rate	Function
270,000	243,000	90%	BusyLoop()
270,000	216,000	80%	BusyLoop()
270,000	189,000	70%	BusyLoop()
270,000	162,000	60%	BusyLoop()
270,000	135,000	50%	BusyLoop()
270,000	108,000	40%	BusyLoop()
270,000	81,000	30%	BusyLoop()
270,000	54,000	20%	BusyLoop()
270,000	27,000	10%	BusyLoop()

Table 6: Resource List for Threads 2-6.

We are also unique in providing practical control of QOS policy decisions, even in the face of a dynamic task set with firm real-time requirements. Our system truly separates policy decisions from accidents of timing, task creation order, and other inadvertent influences. Even with a dynamic task set, we provide exactly the policy that is desired.

This system has a low run-time cost, and provides reliable QOS, and guaranteed scheduling. It guarantees liveness for conventional tasks, and also supports the Quiescent task model for tasks that cannot be denied service.

There remain areas for additional research. First is the better integration of more resources. Our implementation supports the CPU, the FFU, and the Data Streamer. However, we do not specifically manage

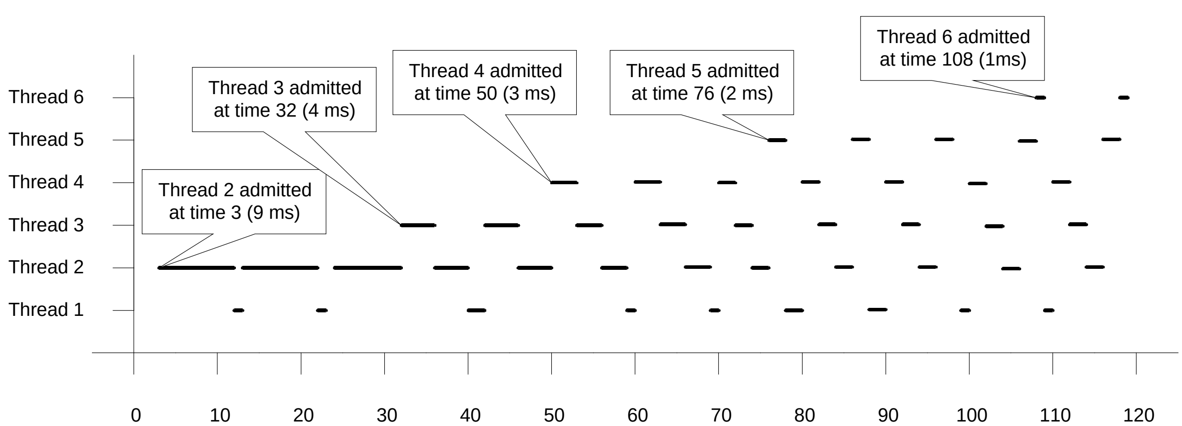

Figure 5: Schedule for Five Load-Shedding Threads Plus the Sporadic Server.

bandwidth as a resource, but we will need to do so when the number of applications using the Data Streamer increases. It is possible that memory should also be incorporated. However, every additional resource increases the complexity of the algorithms.

A second area for more investigation is the Policy Box. Ours can be accessed by applications, the user, and the operating system. There are open issues around policy modifications: when is it reasonable to change the Policy Box, and when should the modification(s) occur to avoid affecting current scheduling guarantees?

Currently, policies are specified as relative rates. There are two problems with this. First, other resources are not integrated. Second, the correlation between the "rates" of the Policy Box are not theoretically well matched to the rates of the applications. Future research will find a better way of expressing possibilities in the Policy Box without limiting the range of resources used by the applications.

8 Acknowledgements

I wish to thank my editor, Sandy Kaplan, my shepherd, Sape Mullender, and the anonymous reviewers who made many helpful comments on earlier drafts. Thanks to my colleagues at Equator Technologies, Inc. who provided endless support for this work.

Bibliography

[Basoglu et al 99] Chris Basoglu, Robert Gove, Keiji Kojima, and John O'Donnell. Single-Chip Processor Media Applications: The MAP1000™. In *Int J Imaging Syst Technol*, 10, 1999, in press.

[Helander & Forin. 98] Johannes Helander and Alessandro Forin. MMLite: A Highly Componentized System Architecture. In *Proceedings of the Eighth ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, Sintra, Portugal, Sep. 1998.

[ISO 96] ISO/IEC International Standard 13818-1, Information Technology – Generic coding of moving pictures and associated audio information: *Systems*. 1996.

[Jeffay & Bennet. 95] Kevin Jeffay and David Bennett. A Rate-Based Execution Abstraction For Multimedia Computing. In *Proceedings of the Fifth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, April, 1995.

[Jones et al. 95] Michael B. Jones, Paul J. Leach, Richard P. Draves, Joseph S. Barrera, III. Modular Real-Time Resource Management in the Rialto Operating System. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, pp. 12-17. IEEE Computer Society, WA, May, 1995.

[Jones et al. 96] Michael B. Jones, Joseph S. Barrera, III, Alessandro Forin, Paul J. Leach, Daniela Rosu, Marcel-Catalin Rosu. An Overview of the Rialto Real-Time Architecture. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, pp. 249-256, Sep. 1996.

[Jones et al. 97] Michael B. Jones, Daniela Rosu, Marcel-Catalin Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, Oct. 1997.

[Liu & Layland 73] C.L.Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. In *Journal of the ACM*, vol.20, pp. 46-61, Jan. 1973.

[Mercer et al. 94] Clifford W. Mercer, Stefan Savage, Hideyuki Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[Nieh & Lam 97] Jason Nieh and Monica S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, Oct. 1997.

[Nieh & Lam 96] Jason Nieh and Monica S. Lam. The Design of SMART: A Scheduler for Multimedia Applications. Technical Report CSL-TR-96-697, Compute Systems Laboratory, Stanford University, June 1996.

[Sprunt et al. 89] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic Task Scheduling for Hard-Real-Time Systems. In *The Journal of Real Time Systems 1*, 1 Nov. 1989.