The Design and Implementation of Real-Time Schedulers in RED-Linux

KWEI-JAY LIN, SENIOR MEMBER, IEEE AND YU-CHUNG WANG

Invited Paper

Researchers in the real-time system community have designed and studied many advanced scheduling algorithms. However, most of these algorithms have not been implemented since it is very difficult to support new scheduling algorithms on most operating systems. To address this problem, we enhance the scheduling mechanism in Linux to provide a flexible scheduling framework. In the real-time and embedded Linux (RED-Linux) project, we implement a general scheduling framework which divides the system scheduler into two components: dispatcher and allocator. The dispatcher provides the mechanism of system scheduling and resides in the kernel space. The allocator is used to define the scheduling policy and implemented as a user space function. This framework allows users to implement application-specific schedulers in the user space which is easy to program and to debug. The framework also relieves the deficiency from the stock Linux scheduler which is not designed for real-time applications. To further enhance its power, a hierarchical scheduling mechanism has been provided in RED-Linux to allow a system designer to integrate different real-time applications together. Using scheduling groups, real-time jobs can be managed and scheduled in a hierarchical manner. In this paper, we discuss how the group mechanism is implemented in RED-Linux.

Keywords*—*Allocator, dispatcher, hierarchical scheduling, Linux kernel, priority-driven scheduling, real-time operating systems, real-time schedulers, scheduling group.

I. INTRODUCTION

Many applications such as avionics systems, traffic control systems, automated factory systems, and military systems require real-time computation and communication services. In real-time systems, all real-time jobs are defined by their timing specifications, such as periodic or sporadic, deadline, response time, etc. Many real-time systems are *hard* real-time systems if missing a deadline may cause a computation to be useless or counterproductive. Some mission-critical real-time systems may suffer irreparable damages if a deadline is missed. It is the system builder's responsibility to choose an operating system that can support and schedule these jobs according to their timing specifications so that no deadline will be missed.

On the other hand, popular applications such as streaming audio/video and multiplayer games also have timing constraints and require performance guarantees from the underlying operating system. Such applications fall under the category of *soft* real-time applications. The application output provided to users is optimized by meeting the maximum number of real-time constraints (e.g., deadlines). But unlike hard real-time applications, occasional violations of these constraints may not result in a useless execution of the application or catastrophic consequences. Nevertheless, a flexible OS support is still needed to manage system resources so that real-time applications can produce the best performance.

Advances in computer technology have also dramatically changed the design of many real-time controller devices that are being used on a daily basis. Many traditional mechanical controllers have been gradually replaced by digital chips that are much cheaper and more powerful. In fact, we believe that the computing power of future embedded digital controllers will be at the same level as that in today's big system servers. As a result, future embedded devices must be able to handle complex application requirements, real-time or otherwise. How we can design real-time operating systems (RTOSs) to support applications with mixed real-time and nonreal-time performance requirements will be an important issue.

These three types of timing requirements (hard real-time, soft real-time, and mixed real-time) are all important for many real-time systems. It is the goal of our research to support many different requirements using one RTOS. In

Manuscript received August 31, 2002; revised February 25, 2003. This work was supported in part by the National Science Foundation under Contract CCR-9901697.

K.-J. Lin is with the Department of Electrical Engineering and Computer Science, University of California, Irvine, CA 92697-2625 USA (e-mail: klin@uci.edu)

Y.-C. Wang was with the Department of Electrical Engineering and Computer Science, University of California, Irvine, CA 92697-2625 USA. He is now with RedSonic Inc., Santa Ana, CA 92705 USA (e-mail: ywang@redsonic.com).

Digital Object Identifier 10.1109/JPROC.2003.814612

this paper, we present our work on real-time and embedded Linux (RED-Linux) and show how it achieves this goal.

A. RTOSs

In this paper, we refer to each schedulable entity in the kernel as a *job*. A job may be a process or a thread in the Linux kernel, depending on whether it shares its memory space with others or not. To meet timing constraints in real-time jobs, it is desirable to have time-cognizant kernel support so that applications may receive the computing resource needed in a timely manner. Although many commercial embedded operating systems (RTOSs) are available on the market, most of these RTOSs have primitive real-time capabilities that are adequate for only very simple real-time applications. On the other hand, future real-time systems often require sophisticated scheduling capabilities and network support to handle applications that may have dynamic workloads and strict performance requirements. They need a new generation of kernel support using innovative yet well-studied scheduling and resource management algorithms.

Most existing operating systems, including Linux, are designed to be nonreal-time operating systems. Their performance goals are as follows.

1) *Fair:* All jobs should be treated fairly, in that they can share all resources in the system in a fair manner.
2) *Efficient:* Operating systems should use resource management policies that are optimized for throughput and to achieve the shortest average response time for all jobs.
3) *User-friendly:* Operating systems should be easy to use and must prevent users from making accidental mistakes.

For real-time applications, however, fairness and efficiency are not as important as meeting real-time constraints. In fact, real-time systems may prefer unfair schedules in order to ensure that urgent jobs can get the resources they need in time to finish executions. For this reason, we need different design goals and primitives for RTOSs.

B. Overview of RED-Linux

Linux has become one of the most popular operating systems in the relatively short life since its creation. There are now millions of Linux installations on different hardware platforms. Due to its open source policy, the Linux community has ported many useful Unix applications and system tools to Linux and made it one of the most capable server operating systems. Commercial support for Linux-based applications is also becoming mature. All these factors make Linux an attractive OS choice for many practical applications.

However, Linux was designed to be a general purpose nonpreemptible kernel. Although newer Linux versions have a limited POSIX-compliant real-time support that allows an application to be defined as a real-time job, current real-time support in Linux is not sufficient. It simply gives real-time jobs higher priorities than normal (i.e., nonreal-time) jobs. No timing constraints such as deadlines, budgets, or periods can be defined in Linux. Moreover, Linux cannot guarantee the response time of real-time jobs. The executions of real-time jobs may be delayed unexpectedly by interrupts or system services (e.g., network protocol processing). Many other events may also block the executions of real-time jobs and cause priority inversions [41].

In this research, we extend the Linux kernel so that it may be used as a real-time and embedded operating system. We have designed and implemented RED-Linux. RED-Linux is designed to support systems that have mixed real-time and nonreal-time jobs with different performance requirements. In particular, the primary contributions of the RED-Linux project include the following.

1) We have implemented a simple and flexible kernel scheduling framework for supporting different real-time scheduling paradigms and policies. The scheduling framework allows each system to define its own scheduling policy in user space. A system designer may not only use existing real-time schedulers, but also design application-specific scheduling policies according to the application requirements.
2) We provide a solution to the kernel preemption issues and significantly improve the system timer resolution. We insert preemption points in the kernel to reduce kernel preemption delay. A microtimer implementation is also adopted to precisely trigger timing events specified with a resolution of microsecond.
3) We have implemented a hierarchical real-time scheduling mechanism. Using hierarchical real-time schedulers, systems can control the resource usage for different job groups and make subsystems independently manageable. The hierarchical scheduling mechanism allows many scheduling policies to be easily integrated.

C. Paper Organization

The remainder of the paper is organized as follows. In Section II, we review some background on RTOSs. In Section III, we discuss how Linux kernel latency can be reduced and controlled. The RED-Linux general scheduling framework is presented in Section IV, which covers the detailed design of the framework. In Section V, we discuss the mechanism to support hierarchical schedulers. We report RED-Linux performance data in Section VI. Some related projects are reviewed in Section VII.

II. RTOS Issues

In this section, we discuss several real-time OS issues, such as scheduling support and kernel latency.

A RTOS must provide a predictable behavior for all real-time applications. One of the most important components is the real-time job scheduler. Compared to schedulers used in nonreal-time or time-sharing systems, a real-time scheduler must meet the timing requirements of every real-time job, not just be fair to all jobs. To date, many real-time scheduling paradigms and algorithms have been proposed and studied. Each of them has its own merits for

meeting some specific performance requirements. However, no one algorithm can be used to meet all requirements. Therefore, it is best if an operating system can facilitate many different scheduling algorithms with little effort. Before we present the RED-Linux scheduling framework in this paper, we first review some well-known scheduling paradigms in Section II-A. These paradigms are all supported by the RED-Linux scheduling framework.

With a powerful kernel scheduler, a real-time kernel still needs to minimize operating system overheads when executing real-time jobs. The operating system latency must be considered as part of a job's execution time when we schedule real-time jobs. In order to avoid missing any deadline, we must be able to control the worst-case operating system latency during job executions. This is discussed in Section II-B.

A. Real-Time Scheduling

In Linux, jobs are scheduled using the time-sharing approach but assigned with different priorities [34]. The priorities are dynamic, i.e., they may be changed at run time. The scheduler keeps track of the status of all jobs and adjusts their priorities to even out their resource consumptions. Long running jobs are given lower priorities than those just entering the system. Input/output (I/O)-bound jobs are given higher priorities than CPU-bound ones. Jobs may change their scheduling state or parameters by making specific system calls such as *sched_yield( )* and *sched_setparam( )*. On the other hand, no specific real-time support is available from Linux other than assigning a special, higher class of priorities for real-time jobs. This only makes real-time jobs run faster since they are scheduled before all nonreal-time jobs. But all real-time jobs are scheduled based on priority. No special provision is given to differentiate other timing properties. In other words, no timing attribute can be defined in the kernel.

Three popular scheduling paradigms have been proposed to schedule real-time jobs, namely, *time-driven*, *priority-driven*, and *share-driven*. These scheduling paradigms are reviewed in this section.

1) Time-Driven Scheduling: In the time-driven scheduling paradigm, the time instances when each job starts, suspends, resumes, and terminates are precalculated and enforced by the scheduler. These instances are chosen *a priori* before the system begins execution. It is therefore easy to predict and to control job behaviors. This paradigm of scheduling is referred as *time-driven* (or clock-driven). Typically, in a system using the time-driven (TD) scheduling, the parameters of hard real-time jobs are fixed and known. A schedule table of the jobs is computed offline and stored for use at run time. The scheduler prioritizes jobs according to this table at each scheduling decision time. In this way, scheduling overhead during run time can be minimized.

For systems with steady and well-known input data streams or events, time-driven scheduling can be used to provide a predictable processing time for each data stream [16], [22]. Applications such as small embedded systems, automated process control, and sensors can be implemented efficiently using this scheduling paradigm. Another feature of TD scheduling is the low jitter property since all executions are started at precise times and produce results at predictable time instances. For systems with close human interactions, such as robots, intelligent devices and automated environments, predictable and jitter-free executions are desirable and often necessary for human users. On the other hand, TD systems cannot facilitate aperiodic jobs easily due to its fixed schedule.

2) Priority-Driven Scheduling: Many recently designed and implemented real-time systems adopt the periodic job model and use the *priority-driven* (PD) scheduling paradigm. Priority-driven scheduling is implemented by assigning priorities to jobs. At any scheduling decision time, the job with the highest priority is selected for execution. PD scheduling includes two classes of algorithms, *fixed priority* and *dynamic priority*. The most well-known fixed priority algorithm is the *rate-monotonic* (RM) algorithm [23], which assigns jobs' priorities according to their period lengths. For dynamic priority scheduling, the most often used algorithm is the *earliest deadline first* (EDF) algorithm [23], which assigns the highest priority to the job with the earliest deadline. Aperiodic and nonreal-time jobs are typically scheduled either as background jobs or served by an aperiodic server job with a well-defined priority and capacity.

However, the PD paradigm allows a misbehaving high priority job to overrun and may reduce the available CPU bandwidth to jobs at lower priority levels. When a system becomes overloaded, the scheduler needs some extra mechanisms to control misbehaving jobs. Thus, in a PD scheduler, it is critical to have a good budget-control mechanism.

Compared to TD scheduling, PD scheduling has several advantages. First, PD schedulers are easy to implement. Many well-known PD algorithms use very simple priority assignments. For these algorithms, the run-time overhead for maintaining a priority queue of ready jobs can be made very small. A TD scheduler requires *a priori* information on the release times and execution times of all jobs in order to decide when to schedule them. In contrast, a PD scheduler does not require this information, making it much better suited for applications with varying time and resource requirements. One of the most important advantages for PD is that there exist simple schedulability checking techniques, so that admission control is easy to implement. In comparison, to insert a new job into an existing TD schedule, all potential conflicts must be carefully inspected and adjusted.

3) Share-Driven Scheduling: Recent advances in computing and communication technologies have led to a proliferation of demanding applications such as streaming audio and video players, multiplayer games, and online virtual worlds. These applications impose only soft real-time constraints, but require a predictable performance from the underlying operating system. Several resource management techniques have been developed for a predictable allocation of processor bandwidth to meet the application needs [4], [6], [9], [40]. Although PD schedulers may be used for such

systems, most PD scheduling algorithms assume a rigid distinction between real-time and nonreal-time jobs and always handle real-time jobs first. Such solutions may not be appropriate. For example, in a real-time video-conferencing system, no hard guarantee on real-time performance are required. For such applications, a *share-driven* (SD) resource allocation paradigm may be more appropriate. The SD scheduling paradigm is based on the general processor sharing (GPS) [32] algorithm and its approximations [17]–[19], [21], [26], [32]. In the context of network packet scheduling, GPS serves packets as if they were in separate logical queues, visiting each nonempty queue in turn and serving an infinitesimally small amount of data from each queue. Using the SD approach for CPU scheduling, every real-time job is requested with a certain CPU share. All jobs in the system share CPU according to the ratio of their requested shares.

Several systems have experimented with share-driven support for real-time networks. In these systems, each application receives a share which, on average, corresponds to a user-specified weight or percentage. Some well-known SD scheduling mechanisms include the weighted fair queueing (WFQ) [11], packet generalized processor sharing (PGPS) [32], WF2Q [7], self-clock fair queuing [13], fair service curves [39], etc. Similar ideas have been used in CPU scheduling such as the total bandwidth server [36], the constant bandwidth server [1], and the constant utilization server [12]. An implementation of the SD scheduler has been reported on the FreeBSD kernel [38], [19].

A special class of SD scheduling is proportionate-fair (pfair) schedulers [3]. A pfair scheduler allows a job to request x_i time units every y_i time quanta and guarantees that over any T quanta, $T > 0$, a continuously running job will receive between $\lfloor (x_i/y_i) \cdot T \rfloor$ and $\lceil (x_i/y_i) \cdot T \rceil$ quanta of service. Pfairness is a strong notion of fairness, since it ensures, at any instant, that no job is more than one quantum away from its due share. Another characteristic of pfairness is that it generalizes the environment containing multiple instances of a resource (e.g., multiprocessor systems). Unlike GPS-fairness, which assumes jobs that can be served in terms of infinitesimally small time quanta, pfairness assumes jobs are allocated with finite durations and thus is a more practical notion of fairness. At present, however, pfair schedulers are not supported in RED-Linux.

B. System Latencies

In Linux, the kernel cannot be preempted when a user job makes a system call to get kernel service. This is because Linux is designed to be nonreentrant. The system cannot preempt a job when it is in the kernel mode. Even if the Linux kernel were designed to be reentrant, it must disable the kernel preemption when executing codes that modify shared data structures. Kernels protect their internal data structures by allowing them to be modified only in critical regions that are guarded by semaphores or spin-locks. Assuming that there is a real-time job waiting to start, and another job is executing in the critical region in the kernel mode, the system cannot transfer the execution to the real-time job until the latter job leaves the critical region. The kernel latency is thus determined by the critical region with the longest execution time. In Linux, the kernel latency can be anywhere between 10 and 30 ms. However, it must be lowered to submilliseconds for most real-time applications.

Another source of unpredictable response time for real-time applications is the overhead involved to handle interrupts. In most operating systems, external devices are allowed to raise interrupts even when a high priority job is running. Regardless of device type, hardware interrupts usually have the highest priority in the system. This may cause real-time jobs to be delayed unpredictably if the number of potential interrupts is not bounded. To solve this problem in Linux, RT-Linux [6] (and RTAI [33]) proposes a software real-time interrupt emulation mechanism. A separate kernel is used under the Linux kernel to intercept all interrupt requests. If an interrupt is defined to be a low latency interrupt, it is handled directly and immediately. Such a software emulation is very effective in most architectures. However, having two kernel layers makes the application design more complicated.

Some recently designed processor (such as ARM) allows interrupts to be raised at different priorities. In the ARM architecture there are two types of interrupts. The IRQ interrupt is used to handle interrupts from normal devices. Another interrupt type, called FIQ, can be used to handle low latency devices. FIQ saves fewer registers than IRQ so that context switching time is reduced. Each of the two interrupt types may be disabled independently. When we disable IRQ in the interrupt service routine, FIQ can still interrupt the system. In this way, we can implement two levels of interrupts. The extra interrupt type provides a convenient solution for low latency devices. As new SOC designs are getting more mature and popular, we believe the hardware low-latency interrupt solution will become more common in the future.

III. Improving Linux Kernel Latencies

We now discuss how the kernel latency issue is addressed in RED-Linux. Three topics are covered in this section: system timer, interrupt handling and kernel preemption. All these affect how precise and how quickly a kernel can respond to an internal event (e.g., time-based trigger) or external event (e.g., pressing a panic button).

A. System Timer

In time-sharing systems, an operating system uses a periodic timer to divide the CPU time among all jobs. By selecting a proper timer frequency to define the time slice, an OS may achieve a good balance between job responsiveness and context switching overhead. Depending on the system architecture, the period of the timer may be somewhere between 1 and 20 ms. However, for many real-time applications, such a resolution is not fine enough. Real-time applications usually need a timer with a microsecond resolution in order to provide a sufficient responsiveness to various devices. To provide a fine timer resolution, yet not to impose extremely high system overhead, a microtimer

design such as UTIME [37] can be used. Rather than using an extremely fine-scale periodic clock to achieve the high resolution, UTIME uses the clock device in the one-shot mode. The clock device will generate a timer interrupt for either the next regular clock interval (typically 10 ms) or the next fine resolution timer event, whichever is earlier. In this way, no excess number of interrupts will be generated, yet all fine resolution timer events can be covered.

We have included the UTIME facility in RED-Linux. In addition, a microsecond timer API has been defined so that users may easily specify timer requests in microseconds.

B. Job Response Time

In addition to timer resolution, a real-time kernel also needs to provide a short job response time. In our discussion, the *job response time* is defined to be the interval between an event occurrence (e.g., device signal, periodic job arrival, etc.) and the start time of job execution in response to the event (e.g., interrupt service, periodic job execution, etc.). It has also been referred to as the *task dispatch latency* in [34]. In general, the job response time includes the following components.

- *Interrupt dispatch time (IDT):* When an interrupt occurs, a system must save all registers and other system execution status before calling the interrupt service routine to handle it.
- *Interrupt service time (IST):* The time used by the interrupt service routine to retrieve information from the hardware device or to gather information from the system.
- *Kernel preemption time (KPT):* The time to preempt the current user job. If the job is running in the user mode, KPT is zero since the preemption may happen immediately. If the user job is running in the kernel mode, KPT is the time before it exits the kernel mode. KPT is discussed further in the next section.
- *Scheduling delay time (SDT):* The time used by the scheduler to select the next user job in response to the interrupt.
- *Context switching time (CST):* The time used to save registers and the status of the current job, and to reset registers and the status of the next job.

Among them, IDT and CST have nearly constant values given a hardware configuration. SDT is determined by the scheduler implementation and, in RED-Linux, depends on which scheduler is used in the kernel. So our discussion in this paper will concentrate on IST and KPT.

IST is the time used to perform the operations for handling an interrupt, such as retrieving data from hardware, adding data to internal data structures, setting up software handlers to process these data at a later time, and setting up hardware for next transition. Therefore, the delay depends on the length of the interrupt handler. For low latency devices, a poorly designed device driver may impose a long IST value and cause an unacceptably long delay for real-time jobs in certain environments. Real-time systems must have carefully designed device drivers to shorten IST.

Another issue is that several interrupts may occur simultaneously. An instance of job response time may contain several back-to-back ISTs plus IDTs. Therefore, the job response time in a real-time kernel may include $N \cdot (\mathrm{IDT} + \mathrm{IST})$, where N is the number of concurrent interrupts. To ensure a low latency, a kernel could impose a hard limit on N. However, this will reduce system flexibility since some restriction will be placed on how users may implement their applications. Therefore, no special provision is used in RED-Linux to place a strict bound on N. We consider this to be an application design issue. Application designers must make sure a system has a reasonable bound.

From the above discussion, we can see that job response time depends on many factors that are implementation dependent. Indeed, the actual performance of a real-time system is an end-to-end performance issue. Having a real-time kernel solves only half of the problem. Systems still must have carefully designed schedulers, device drivers, etc. to make sure all response times are bounded.

C. Kernel Preemption

To reduce job response time, we must also improve the kernel preemption to reduce KPT. Otherwise, a low priority job can block another higher priority job for a long time by staying in the kernel mode.

Two different approaches are possible to preempt a job running in the kernel mode. The first is the *full preemption* model and the other is the *cooperative preemption* model. The implementation of a fully preemptive (FP) kernel is similar to that of a symmetric multiprocessor (SMP) kernel. In an SMP kernel, two or more CPUs share the same kernel data space. We need to allow for more than one job running in the kernel mode. Therefore, all accesses to kernel data structures need to be protected by locking semaphores or spin-locks.

Another approach of providing preemption in kernel is to insert *preemption points* in the kernel code. A preemption point has a code structure as follows:

```
if (need_preempt) {
 call scheduler to yield CPU
}
```

The condition "`need_preempt`" is set whenever a real-time job (with a higher priority than the current running job) is waiting to be executed. When a kernel execution reaches a preemption point, it will check `need_preempt`. If any high priority real-time job is waiting, the kernel may voluntarily suspend its execution. On the other hand, if there is no high-priority job waiting, the kernel will continue its normal execution.

Fig. 1 illustrates the difference between the FP kernel and the preemption point approach. The left hand side shows the execution of a FP kernel, while the right hand side shows a preemption point kernel. The gray areas are critical regions that are protected by "lock" and "unlock" operations in the FP kernel. In a kernel with preemption points, kernel execution cannot be preempted except at preemption points. As

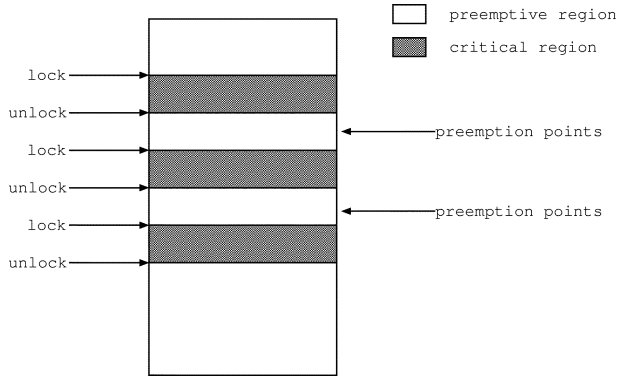


Fig. 1. Fully preemptive kernel versus preemption points.

long as the preemption points are outside of critical regions, no sharing conflict is possible.

Both approaches may achieve a reasonably small preemption delay. With preemption points, the preemption delay is the longest distance between any two consecutive preemption points. On the other hand, the execution overhead with preemption points is usually smaller than a FP kernel since it has fewer semaphore locks. The resolution of the FP kernel is determined by the longest execution time on any critical region.

In RED-Linux, preemption points have been added to the kernel. Originally, we developed our own kernel tool to measure the execution latency for kernel functions. Preemption points are then inserted to those functions with a long latency. Our experience shows that the preemption delay is greatly improved compared with the original Linux kernel. More on the performance of preemption points can be found in [43]. The current version of RED-Linux has adopted the low-latency patch produced by Andrew Morton [30]. It also should be mentioned that a *preemption patch* has been accepted in Linux Version 2.5.4. The preemption patch implements a FP Linux kernel. However, a recent study [2] has shown that the preemption patch is not consistently better than the low-latency patch. We will consider both patches and others (such as [25]) in future RED-Linux versions.

IV. Flexible Scheduling

In this section, we present the *general scheduling framework* (GSF) to implement different scheduling algorithms in RED-Linux [44]. In this framework, applications can have their own schedulers implemented using the GSF API. This approach is very similar to that used in many microkernels. In Mach [8], only some basic primitives are implemented in the kernel. Other high-level services such as file system and interprocess communication are implemented as servers in user space so that users may change from one implementation to another. Users can even execute multiple servers in order to execute different types of applications. This is our goal in the design of RED-Linux for supporting real-time jobs.

A. Basics of RED-Linux Scheduling

In our model, a job is the smallest execution unit to be managed by a scheduler. A scheduler selects one of the jobs to execute in the next CPU time slot according to the scheduling policy used. When one of a job's resource constraints is used up, the job will be discarded. A periodic job will be ready to execute at the beginning of every period, while an aperiodic job is ready when a new event arrives or at the time defined by the aperiodic server algorithm. Therefore, a periodic application can be thought of as a steady stream of jobs and an aperiodic application can be thought of as a set of irregular jobs.

A scheduler is responsible for assigning system resources to every job in the system according to some performance requirements. Usually, the high-level performance specification defined by applications cannot be used by a kernel scheduler directly. The performance specification must be translated into low-level parameters that are acceptable to the mechanism of the scheduler. There needs to be a middle layer between the low-level scheduler and the user application to translate the high-level performance parameters into the low-level scheduling parameters.

For example, in a rate monotonic scheduler, applications define periods and execution times for each job. We must translate every job's period and execution time into start time, finish time, and priority so that the kernel scheduler can make scheduling decisions. Traditionally, this middle layer is integrated with the low-level kernel scheduler. However, there are benefits if we separate these two modules in a traditional scheduler.

1) It is more difficult to redesign a low-level scheduler than a performance parameter translator, because the low-level scheduler is tightly coupled with the structure and functionality of the kernel. If we separate these two modules, we can easily redesign a new performance parameter translator and to obtain a new scheduler that uses the same low-level scheduler. It simplifies the procedure of implementing new scheduler algorithms.
2) If the performance parameter translator is separated from the low-level scheduler, we can let the application determine the scheduling policy to be used. An application can even modify a scheduler on the fly based on some dynamic run-time conditions. Such a flexibility is not available in traditional kernel designs.

B. Scheduling Parameters

Different scheduling paradigms use different attributes to make scheduling decisions. In order for all major scheduling paradigms to be supported in our framework, it is important that all useful timing attributes be included in the framework so that they can be specified by schedulers. On the other hand, we want to define only a minimum set of basic timing attributes. Therefore, the attributes should be orthogonal to each other. We have identified the following four basic scheduling attributes for each job.

Priority: A job's priority defines the importance of the job relative to other jobs in the system.

Start time: The start time defines when a job becomes eligible for execution. A job cannot be executed before

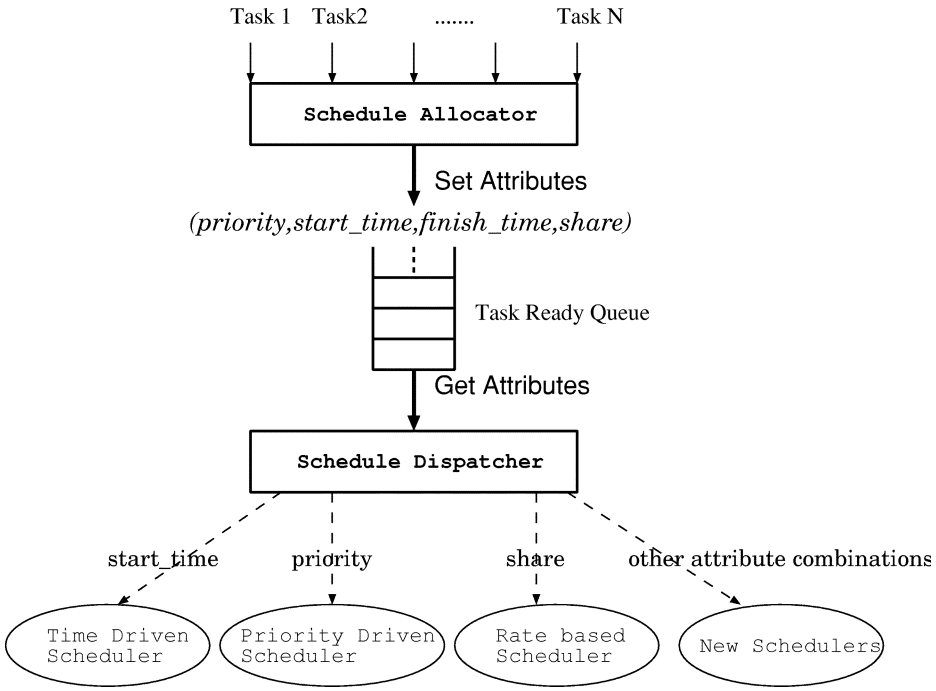


Fig. 2. GSF framework.

its start time. If a job arrives earlier than its start time, it is delayed until its start time.

Finish time: The finish time of a job defines when the job is no longer eligible for execution. At a job's finish time, even if the job has not been finished, its execution must be suspended or aborted.

Budget: The budget is the allowed execution time of a job. A job will be suspended when the execution time it has used exceeds its assigned budget.

Among the four, the start time and the finish time combined specify the *eligible interval* of a job execution. The budget specifies the remaining execution time of a job. These attributes can be used as constraints. However, each of the timing attributes can also be used as the scheduling parameter when a scheduler needs to select a job for execution. One can also use an objective function to integrate some of these attributes.

C. Scheduler Components

We propose a general scheduling framework (GSF) in RED-Linux. GSF is a two-component scheduling framework (Fig. 2) that separates the low-level scheduler, called the *dispatcher*, which renders the CPU scheduling mechanism, from the performance parameters translator, called the *allocator*, which implements the scheduling policy chosen. We also propose a method to efficiently exchange scheduling data between these two components.

In the GSF, it is the allocator that sets up the four real-time scheduling attributes associated with each job according to the selected scheduling policy. When a real-time application arrives at a system, the allocator assigns scheduling attributes to all of its jobs based on application timing requirements such as period, interarrival time, execution time, requested share, deadline, etc. based on the current scheduling policy designated by the application. In other words, the allocator transforms the timing requirements of an application into scheduling attributes to be used by the dispatcher. In addition to assigning attribute values, the allocator also specifies the relative evaluation order of scheduling attributes, since each job has multiple scheduling attributes. The allocator may choose one of the attributes as the primary attribute so that the dispatcher will follow a specific scheduling discipline. For example, if the *priority* attribute is designated as the primary attribute, the dispatcher becomes a priority-driven scheduler. If the *finish_time* attribute is used as the primary one, the dispatcher acts as an EDF scheduler. By default, the dispatcher will use the priority attribute as the only scheduling parameter, and other attributes as constraints.

After each job is assigned its scheduling attributes by the allocator, it is sent to the ready queue maintained by the dispatcher. The dispatcher scans through all eligible ready jobs from the ready queue and chooses one job for execution. A timer is used to enforce each job's budget so that no job will use more than its assigned budget. Every time after the current job finishes or runs out of its budget, the dispatcher will choose the next one to execute. If a job finishes its execution or misses its deadline, its entry in the ready queue will be deleted by the dispatcher. If a job is preempted by a newly arrived job, its budget will be reduced by the time already used.

The way in which the allocator sets up job attributes and the dispatcher selects jobs for execution together implements the desired scheduling policy. In our design, the dispatcher always runs in the kernel space for performance reasons. But the allocator may run in the user space or kernel space. If the allocator does not need to use application-specific information that cannot be collected in the kernel space, an allocator can run in either space. Otherwise, the allocator must

run in the user space. For example, some standard allocators for RM, EDF, or TD have been written as kernel modules. If an application wants to use this scheduler design, it can be placed in the kernel space at run time to reduce the context switching overhead between the allocator and the dispatcher.

GSF is able to accommodate many existing scheduling disciplines. System developers may choose different scheduling disciplines tailored to their specific application environment. The dispatcher may even switch between different scheduling disciplines at run time.

D. Implementing GSF

1) Dispatcher: The dispatcher is implemented as a kernel module in Linux. The dispatcher only schedules those real-time jobs registered at the allocator. All non-real-time jobs are managed by the original Linux scheduler. The dispatcher is used to determine the execution order just like a traditional kernel scheduler. The scheduling attributes of a job in the dispatcher will not change during the lifetime of the job.

In the job queue of the dispatcher, we may find several jobs that are generated by the same application (as in the case of a periodic task). The lifetime of each job is much shorter than that of the application. Usually, a job's lifetime is equal to the individual period of a periodic application.

For each job, three timer events are defined to control its execution: startup, budget, and finish. A startup timer is used to move a job into the running queue. A budget timer is used to suspend a job when the job's budget is used up. Finally, a finish timer is used to move a job out of the running queue when a job has reached its finish time. The values of the start and finish timers are determined by the start time and the finish time attributes of the job, respectively. The budget timer is decided dynamically at run time. Whenever the dispatcher is invoked, the dispatcher will select a job from the running queue to be the next CPU job and update its budget timer at the same time. The budget timer is reset to the current time plus the job's remaining budget. A job can (and should) call `GSI_abort_job( )` to relinquish its remaining budget whenever it has finished its execution and is to be removed from the running queue. Another reason for any unused budget to be abandoned is when the job becomes *ineligible*, i.e., it has already past the job's finish time.

The timer resolution depends on the platform. For efficiency reasons, we will force a timer to expire immediately if the expiration time is within one time slice away from the current time so that we can avoid unnecessary context switches. The actual time slice granularity is some tens of microseconds on most platforms.

There are three events that will cause a dispatcher to be invoked. First, it will be invoked when any timer has expired. Second, whenever a `GSI_abort_job( )` is called. In this case, the dispatcher will select the next job from the running queue. Finally, when the allocator sends a new job to the dispatcher, the dispatcher will be invoked to calculate the job's effective scheduling parameter.

2) Allocator: The responsibility of the allocator is to determine the performance parameters of all jobs according to the current state of jobs. After a new job is sent to the dispatcher, the dispatcher will schedule it according to its primitive scheduling parameters. In this section, we will discuss several design issues of the allocator.

The allocator could be part of the application or middleware. In the latter case, the allocator itself is also a real-time job. It must schedule itself in order to receive CPU time to process the information sent by the dispatcher. It is the responsibility of the allocator to assign itself enough time to make scheduling decisions on time. Usually, the allocator should assign itself the highest priority in the system so that it will not be blocked by any job in the system. The allocator may also use the system idle time to make scheduling decisions so that it will not delay the execution of other real-time jobs.

Factors that influence scheduling decisions include performance parameters of the application, the executing status, and the number of times a job has executed. For some scheduler policies, the actual status of job executions is irrelevant. For example, a rate-monotonic scheduler only needs to send a new job into the dispatcher at the beginning of every period. On the other hand, many newly designed algorithms make use of the execution history. For example, if we want to implement a scheduler that does not know the job execution time in advance, we may want to adjust its execution budget at run time adaptively. For implementing these types of schedulers, we need to monitor the actual execution time of every job at run time. If a job terminates before it exhausts its preassigned budget, we can reduce its budget so that more jobs can be admitted to the system. Conversely, if a job is always preempted due to insufficient budget, we should assign it more resources so it may complete its execution. To get feedback events from the dispatcher, two mode switches between user mode and kernel mode are needed. To send a new job to the dispatcher, two additional mode switches are required. Therefore, there will be additional overheads of four mode switches. The tradeoff between scheduler flexibility and efficiency is an important consideration for system designers.

E. Using GSF to Implement Scheduling Algorithms

One of our goals in designing GSF is to facilitate the implementation of existing scheduler algorithms. Different scheduling algorithms require the allocator to update the scheduling attributes in a different way. The assignment of the attributes for these schedulers are shown in Table 1. The start time and finish time of the TD policy should be set up according to the time table produced by the TD scheduler. Most of the share driven schedulers will schedule all jobs according to their *virtual finish time*, which is used as its priority value. The finish time attribute for each job using SD scheduling is usually set to ∞ since it is eligible for execution until it uses up its budget.

In addition to implementing the basic scheduling paradigms, our framework can also be used to implement more complex schedulers easily. Since the allocator updates scheduling attributes dynamically, we can implement many dynamic schedulers in GSF. In this section, we show the

Table 1
Attributes for Different Scheduling Policies

Policy	Start time	Finish time	Budget	Priority	Scheduling parameter
RM	BP	EP	B	1/P	priority
EDF	BP	EP	B	NA	finish time
Time Driven	scheduler	scheduler	scheduler	NA	start time
Share Driven	arrival time	infinity	B	virtual finish time	priority

BP: the beginning of the period. *EP:* the end of the period.
P: the period of the job. *B:* the budget of the job.
"*Scheduler*" means the value is determined off-line by the scheduler.

allocator implementations of two well-known scheduling protocols: imprecise computation [24] and dual priority [10]. In both examples, the actual program to be executed by user jobs is application-dependent. These jobs may be created by using the `fork( )` command in the allocator and then invoking the actual application code. Another way is to create a job independently and then pass its `PID` to the allocator and the dispatcher. The latter approach is more common when a general-purpose allocator (designed for a well-known scheduling policy) is adopted in the system.

1) Implementation of Imprecise Computation: The imprecise computation model [24] can be used to structure real-time systems that may have insufficient computing resources for all applications. In many real-time applications, the *mandatory* part of an application must be completed by a specific deadline, while its *optional* part is executed only after the system can meet the demand of all mandatory executions. Otherwise, some optional parts may be skipped.

Using the RED-Linux scheduling framework, we set the priority to be in the high band when one application is in its mandatory part. When the application finishes the mandatory part and goes into the optional part, we will set the priority to the low band. Therefore, the optional part is executed only when there is no other high priority job waiting to be executed. An example implementation is listed in Figs. 3 and 4. In Figs. 3, the allocator first registers itself to the RED-Linux kernel. It then defines the task scheduling attribute values. It also uses a status variable `isOptional` to denote whether the current job is mandatory or optional. After sending the first mandatory job to the dispatcher, it enters a loop waiting for the next event and then invokes the Imprecise_EventHandler( ) function (Fig. 4). The Imprecise_EventHandler( ) regenerates mandatory and optional jobs alternatively, setting their attributes according to the last execution. At any time, only one job is active and is sent by the allocator to the dispatcher. When the job is finished, it attribute values will be updated by the Imprecise_EventHandler( ). Since the attribute update is one job at a time, no data sharing locking is necessary.

In this example (Fig. 3), the scheduling policy for the whole system uses only the priority value as the primary scheduling attribute by setting `qos[0]` to `QOS_PRIORITY`. If there are other secondary attributes, `qos[1..3]` can be set to other values. A -1 value means no additional secondary attribute is used. If no value is set in any of the `qos[0..3]` array, the priority attribute is used as the primary attribute by default.

2) Implementation of Dual Priority Scheme: The aim of the dual priority scheduling [10] is to provide a more responsive execution for nonreal-time jobs by running all real-time jobs only when there is no best-effort job to execute, or as late as possible without violating real-time jobs' deadlines if there are nonreal-time jobs present. Using the dual priority scheduling, real-time jobs execute at either an upper or lower band priority level. Upon release, each real-time job assumes its lower band priority. At a fixed time from the release, which is called the *promotion time*, the priority of the job is promoted to the upper band. At run time, other nonreal-time jobs are assigned priorities in the middle band.

Using GSF, the allocator adds two jobs to the dispatcher queue when a real-time job arrives. One job is given the lower band priority with the beginning of the period as its start time, and the finish time is set to the promotion time. The other job is given the higher band priority with the start time equals to the promotion time and the finish time set to the end of the period. The two jobs collectively can use the total budget as defined by the job execution. In this way, the dual priority scheme can be implemented. Fig. 5 shows only the event handler. The dual priority allocator is very similar to Fig. 3.

The high-priority job must update the budget attribute according to the execution status of the low-priority job. For example, if a low-priority job uses one-third of its budget, the remaining budget value is kept in the event attribute `e.budget`. This adjustment should be done by the allocator.

V. Group Scheduling

The general scheduling framework addresses the requirements of supporting different scheduling paradigms. With the framework, many powerful schedulers can be easily implemented. As real-time systems get more complex, different parts of a real-time system may be used to meet different needs from applications. A subset of the jobs may need to share some computing or data resources with each other closely. Therefore, schedulers may divide jobs into groups that are semantically related. In this way, a scheduler first decides which group is to be executed and then which job in the group. This is the idea behind hierarchical schedulers. Using hierarchical schedulers, we can control the resource budget for different groups and, thus, different functionalities of the system. In this section, we describe the support for group scheduling in RED-Linux [45], [47].

```
// Allocator for Imprecise Computation
typedef struct {
  int period;         // period of the job
  int budget;         // budget of the job
  int isOptional;     // mandatory or optional
  GSI_JOB job;        // scheduling attributes of the job
} TASK;

main()
{
  TASK t;
  GSI_JOB *j;
  GSI_QOS qos[4];     // scheduling policy selector
  GSI_Event e;        // dispatcher event feedback

  GSI_register();                    // Register to get the allocator privilege
  memset(&t,0,sizeof(t));           // Clean up the TASK data structure t
  t.period = GSI_msec_to_ticks(10);  // Period is set to 10ms
  t.budget = GSI_msec_to_ticks(3);   // Budget is set to 3ms
  t.isOptional=0;                    // First job is a mandatory part

  j=&t.job;                        // Set up the first job
  j->starttime = GSI_now() + t->period;
  j->finishtime = j->starttime + t->period;
  j->budget = t->budget;           // Assign the full budget
  j->priority = HIGH_PRIORITY;     // Mandatory part has the high priority
  j->private=(int)&t;              // Pointer to the task data structure
  qos[0] = QOS_PRIORITY;           // Set priority as the primary scheduling attribute
  qos[1] = -1;                     // Define no other scheduling attribute
  GSI_setQOS(qos);                 // Send scheduling policy to the dispatcher
  GSI_addjob(j,1);                // Send one job to the dispatcher
  while(1) {
    int n = GSI_getevents(&e,1);  // Wait until a response event from dispatcher
    Imprecise_EventHandler(&e);   // Call the event handler in Fig 4
  }
}
```

Fig. 3. Imprecise computation allocator.

A. Scheduling Group

In Linux, a *process group* may be defined and is used mainly for multicast. If a message is en to the group, the message is sent to all member processes in the same group. However, processes in a Linux process group are scheduled independently. To support the multilevel hierarchical scheduler in RED-Linux, we introduce the concept of *scheduling group*. A scheduling group contains a set of jobs that are scheduled by a group scheduler using a specific scheduling policy. A scheduling group can be thought of as a set of jobs running on its own virtual processor(s), independent of other scheduling groups. Each scheduling group has a maximum system budget which is to be shared by all jobs in the group.

A *group* number is defined for each job in RED-Linux for scheduling group identification. Every job in a group is assigned the same group number and shares a common scheduling budget assigned to the group. Conceptually, at the highest level (group 0), the kernel scheduler manages all scheduling groups and selects one group for execution according to the system-level scheduling policy. Once a group is selected, the group scheduler then selects a job in the group for execution according to the group level scheduling policy. The scheduling mechanism is defined recursively since any job in a group can be defined as a group itself.

Another attribute, the *server* number, is defined for each *group job* (i.e., a virtual job that is actually a group of jobs). Each group job is associated with a job queue which holds all jobs that will be scheduled using the group job's budget. Normal real-time jobs do not have a server number.

The dispatcher algorithm is defined as follows.

Step 1) The dispatcher selects a job K from all eligible jobs in group 0 (top level) according to the scheduling policy of group 0.

Step 2) The server list J is initialized to NULL.

Step 3) If K is a real-time job, execute it and go to Step 4); else if K is a group job with server number i, select a job L from the eligible jobs in group i.

a) Set a new timer to be

$$\text{timer} = \min \begin{cases} \text{current timer} \\ \text{now} + K.\text{budget} \\ \text{now} + L.\text{budget} \\ K.\text{finishtime} \\ L.\text{finishtime} \end{cases}$$

```
// Event Handler for Imprecise Computation
Imprecise_EventHandler(GSI_Event *e)
{
  TASK *t = (TASK *) e->private;    // Get the task data structure
  GSI_JOB *j = &t->job;

  switch(e->type) {
   case GSI_EV_JOB_EXPIRED:          // The last job has finished
   if (t->isOptional) {
      //If the last job is an optional job, produce a mandatory job
      // for the next period.
      j->starttime += t->period;
      j->finishtime = j->starttime + t->period;
      j->budget = t->budget;       // Reset to the full budget
      j->priority = HIGH_PRIORITY; // Mandatory part has the high priority
   } else {
      //If the last job is a mandatory job, produce an optional job in the same period
      j->budget = e->budget;      // Use only the remaining budget
      j->priority = LOW_PRIORITY; // Optional part has the low priority
   }
   t->isOptional = 1 - t->isOptional;
   GSI_addjob(j,1);               // Send the job to the dispatcher
  }
}
```

Fig. 4. Event handler of imprecise computation.

where *now* is the current time, $K.\mathrm{budget}$ and $L.\mathrm{budget}$ are the budgets of jobs K and L, and $K.\mathrm{finishtime}$ and $L.\mathrm{finishtime}$ are the finish times of jobs K and L.

b) Append the group job K to the server list J.

c) Set $K = L$ and repeat Step 3).

Step 4) When K finishes or is interrupted, deduct the actual execution time consumed by K from the budget of all servers in list J. Go back to Step 1).

In Step 3), when the dispatcher selects a job group, the dispatcher selects a job from the group. The dispatcher will repeat Step 3) if it again selects another group job. In this way, we can implement a hierarchical scheduler.

We now show an example using the scheduling group mechanism. Suppose we want to implement a hierarchical scheduler as in Fig. 6. Assume the time-driven scheduler executes jobs of the RM scheduler in the first half of each cycle and EDF at the second half of each cycle. At the beginning, the allocator creates four real-time jobs in the sporadic mode. The group numbers are 1, 2, 3, 4, for RM, EDF, EDF1, and EDF2 respectively. Therefore, four jobs will be sent to the dispatcher. They are shown in Table 2.

Group 0 uses the time-driven policy. One of the RM or EDF server jobs will be chosen. If the RM server job is selected, the dispatcher will select a real-time job from queue 1. If an EDF server job is selected, it may be one of the server jobs EDF1 and EDF2. Because they are server jobs again, the dispatcher will select from group 3 or group 4 again. Real-time jobs can be in groups 1, 3, and 4.

The group scheduling can be used to implement hierarchical resource management, which is very useful in applications with multiple jobs. For example, we may want to control the resources used by any request received by a web server. For each request, the web server may create several jobs to do different jobs. Typically, we will have a web job, a CGI job, and a database job. By using the hierarchical groups, we can control the resource consumption of a subsystem without understanding how the subsystem allocates and schedules resources among its various independent activities. We simply control the total budget used by the group.

B. Budget Group

In the scheduling group model, the job scheduling and budget sharing among jobs in the same group are defined by the scheduling policy of the group. Jobs that belong to different scheduling groups cannot share a common budget. For example, we cannot share a budget between two jobs, one of which belongs to an EDF scheduling group and the other belongs to an RM scheduling group. This constraint turns out to be quite inconvenient when we try to implement several popular scheduling algorithms.

We thus provide a simpler group concept called *budget group* [46] to facilitate only budget sharing but not group scheduling. A budget group contains jobs that share the total

```
// Event Handler for Dual Priority Scheduling
typedef struct {
  int period;         // period of the task
  int budget;         // budget of the task
  int high_priority; // the priority after the promotion time
  int low_priority;  // the priority before the promotion time
  int promotion;      // the promotion time
  int isPromoted;     // The current priority status of the task
                      //   0 :  before promotion
                      //   1 :  after promotion
} TASK;

DualPriority_EventHandler(GSI_Event *e)
{
  TASK *t = (TASK *) e->private;
  GSI_JOB *j = &t->job;

  switch(e->type) {
   case GSI_EV_JOB_EXPIRED:
    if (t->isPromoted) {
      //If the last job is a high priority job,
      // produce a low-priority job for the next period.
      j->starttime += t->period;
      j->finishtime = j->starttime + t->promotion;
      j->budget = t->budget;
      j->priority = t->low_priority;
    } else {
      //If the last job is a low priority job, produce a prompted job
      // high-priority job.
      j->finishtime = j->starttime + t->period;
      j->budget = e.budget;           // Use only the remaining budget
      j->priority = t->high_priority;
    }
    t->isPromoted = 1 - t->isPromoted;
    GSI_addjob(j,1);                  // Send the job to the dispatcher
  }
}
```

Fig. 5. Event handler of dual priority scheduler.

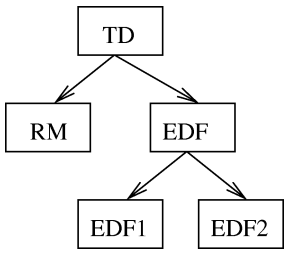


Fig. 6. Hierarchical schedulers.

budget for the group. But jobs may be scheduled by different scheduling policies. A job in a budget group could be a normal job or a group job. Each normal job still has its budget attribute. But if it belongs to a budget group, its execution will be constrained by the group's budget. In turn, the group job needs to check with its parent node on the budget group tree (Fig. 7). The recursive budget checking stops at the top level system job. The available budget for a job is

Table 2
Scheduling Group Attributes

	start time	finish time	priority	budget	group no.	server no.	policy
RM	0	$P_t/2$	-	$P_t/2$	0	1	priority
EDF	$P_t/2$	P_t	-	$P_t/2$	0	2	finish time
EDF1	$P_t/2$	P_t	-	$P_t/4$	2	3	finish time
EDF2	$P_t/2$	P_t	-	$P_t/4$	2	4	finish time

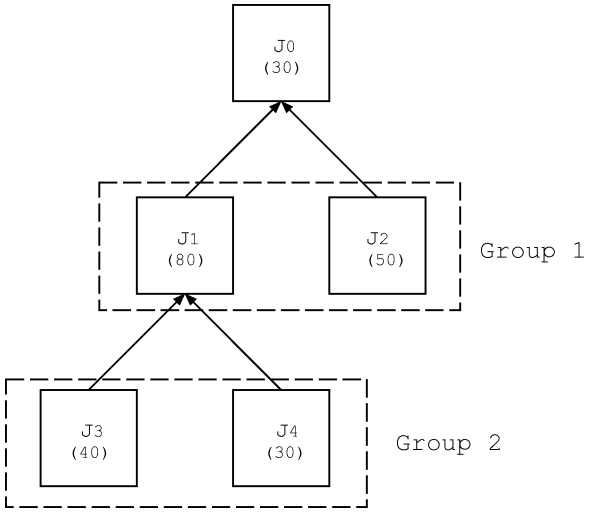


Fig. 7. Budget group tree.

the smallest budget value among all its ancestor nodes on the budget group tree.

Unlike a scheduling group, there is no job selection procedure on these jobs. The budget group is only used to facilitate budget sharing among jobs. Following dispatcher and allocator design philosophy, the way to manage the budget among jobs is decided by the allocator, which implements the policy. In other words, the allocator decides how to structure budget groups and budget sharing among jobs. In the dispatcher, the mechanism is implemented to support budget groups.

1) Budget Group Hierarchy: All budget groups in the dispatcher build a hierarchy, which we call a *budget group tree*. All normal jobs are leaf nodes in the tree, while budget group jobs are intermediate nodes. The example in Fig. 7 shows the budget group hierarchy. Jobs J_1 and J_2 belong to the same budget group, so they share the available budget in budget group job J_0. Job J_1 itself is a budget group job whose budget is shared by two normal jobs J_3 and J_4. The attributes of all jobs (including normal jobs and group jobs) in Fig. 7 are listed in Table 3.

When J_3 is selected by the dispatcher for execution, J_3 has a budget of 40. However, since it is in budget group 2 and shares budget with J_4, it needs to check the total budget available for the group 2. J_1's budget is 80, i.e., the group 2 has budget 80 available. On the other hand, J_1 is in group 1 that is constrained by the budget value of 30 in J_0. Therefore, J_3 cannot receive the budget of 40, but only a budget of 30.

Using this mechanism, it is up to the allocator to assign the budget values of these groups so that the dispatcher can construct and maintain the hierarchical structure accordingly. A budget group job may have several budget segments available at the same time. This is because the allocator will replenish a budget group from time to time, each time with a different start time and finish time (they are, in fact, *available time* and *expiration time*). Depending on the scheduling policy used, these multiple budget segments could overlap. In this case, the dispatcher needs to determine which segment to use. In RED-Linux implementation, EDF is used to select

Table 3
Budget Group Parameters

	priority	budget	group	server
J_0	-	30	0	1
J_1	-	80	1	2
J_2	-	50	1	1
J_3	-	40	2	2
J_4	-	30	2	2

the eligible budget for a job execution, i.e., the dispatcher always chooses the budget segment with the earliest finish time, since it wants to consume the budget that will expire first.

2) Operations Before a Normal Job Execution: When a normal job is selected for execution, it needs to determine the budget available. The algorithm for the dispatcher is defined as follows.

1) Find the budget group it belongs to.
2) Check all budget group nodes on the path from itself to the root by traversing bottom-up to see if there is enough budget available at each level. If we find an intermediate node that does not have any eligible budget available, the normal job is added to the node's sleep queue.
3) For each intermediate budget group, select a budget in the group job's budget queue. If there are multiple budgets available in the queue, use EDF to determine which budget should be used. Any expired budget is removed from the budget queue.
4) When the job finishes, reduce the budget of all its ancestor group jobs by the actual time it consumed.

In Fig. 7, if J_3 is selected, J_3 can only have a budget of 30 since the root J_0 has only a budget of 30. When J_3 finishes, the budget of J_1 and J_0 will be reduced by the actual time used by J_3.

C. Integration of Scheduling Group and Budget Group

Since RED-Linux supports both budget group and scheduling group, three types of jobs may exist in the ready queue of a dispatcher: normal real-time jobs, scheduling group jobs, and budget group jobs. Each job has its start time, finish time and budget. The start time and finish time of each job is defined by the application. We assume the root node belongs to group 0.

Both scheduling group jobs and budget group jobs maintain their own queues. Each group job has two queues. One of them is the running queue of its children. It selects a child job from this running queue according to its scheduling policy. The other is the sleep queue for its children that cannot be selected for execution. For a budget group job, one queue is also the sleep queue that holds its children that do not have any budget. The second queue is the budget queue, because it may have several available budgets. The traversal of the budget group tree is bottom up, i.e., from a leaf node to the root node, while in the scheduling group tree, the traversal is top-down from the root to a leaf node.

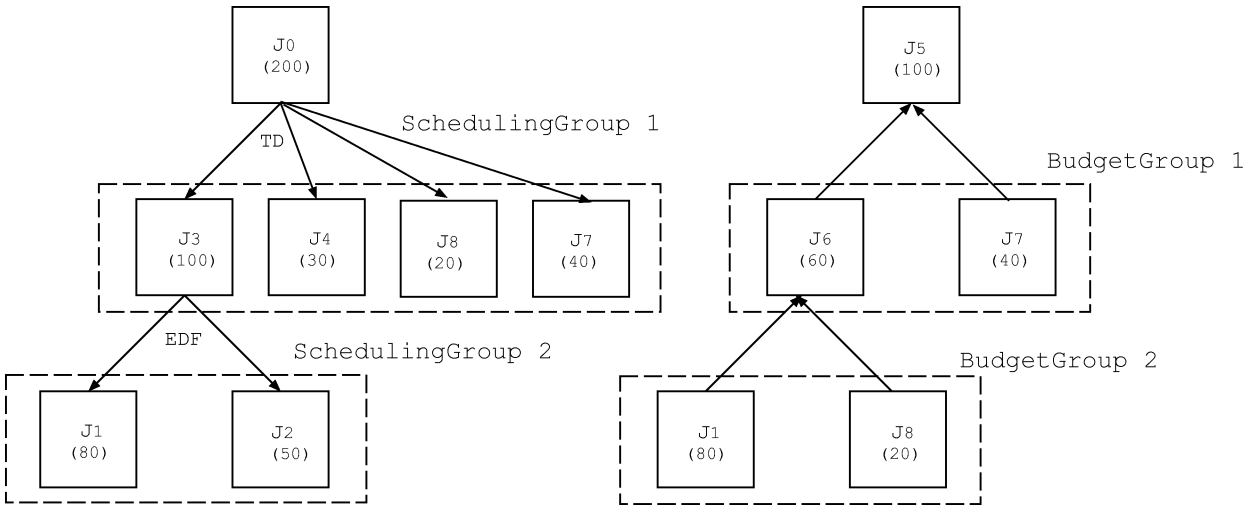


Fig. 8. Integrating scheduling group and budget group.

In Fig. 8, we show an example of both trees. All jobs' budget and group related attributes are shown in Table 4. Let us follow the scheduler operation for the execution of job J_1. The dispatcher will start from J_0 to select a job in scheduling group 1. Using time-driven scheduling, the dispatcher selects group job J_3 according to its start time. Since J_3 is a group job, the dispatcher continues the selection operation to select J_1 using EDF. The budget constraints in the scheduling group tree have not created any problem, since all J_1's ancestors (J_3 and J_0) have enough budget. So J_1 still has a budget of 80 at this point. But we also need to check the budget group tree. Since J_1's parent J_6 has only budget of 60 available, J_1 will be allowed a budget of 60. When checking with J_5, J_1 will maintain the budget of 60 for execution. This is the end of budget checking and J_1 will have a budget of 60. After the execution, all nodes' budget values will be reduced by the actual time used by J_1.

Table 4
All Job Parameters

	budget	sched.group	sched.server	budget.group	budget.server
J_0	200	0	1	-	-
J_1	80	2	2	2	2
J_2	50	2	2	-	-
J_3	100	1	2	-	-
J_4	30	1	1	-	-
J_5	100	-	-	0	1
J_6	60	-	-	1	2
J_7	40	1	1	1	1
J_8	20	1	1	2	2

Table 5
Dispatcher Delays (In Microseconds)

No. Jobs	min delay	average delay	max delay
1	0.24	1.14	22.39
10	0.28	1.27	24.65
100	0.44	4.11	96.59

VI. Performance of RED-Linux

When considering the performance of a real-time scheduling framework, the most often used criterion is the scheduler overhead. One metric for comparing two different implementations of the same scheduling algorithm is the time used to make scheduling decisions. Another important indicator of performance is the job response time, which includes interrupt delay, context switch time, scheduling delay, and others. Different implementations of the same scheduling algorithm may result in different scheduling delays. Therefore, we can use the scheduling delay as another index of performance.

The scheduling delay in our general scheduling framework is the time used by the dispatcher to select the next running job. The time used by the allocator should not be considered as a part of the scheduling delay because the allocator is executed as another user process, not part of the kernel. When the dispatcher selects the next running job, the allocator usually does not need to be involved in this decision. Of course, there are exceptions: if the dispatcher's job queue is empty, the dispatcher's scheduling decision will be delayed until the allocator is notified to send more jobs to the dispatcher. In this case, the time used by the allocator must be considered as part of the scheduling delay. Therefore, the allocator must be designed carefully to prevent or eliminate such situations. If the dispatcher needs to call the allocator in the scheduling decision process, the context switch and mode change overheads between the allocator and the dispatcher will significantly degrade the scheduler performance.

In this section, we measure the overheads for the dispatcher to do job selections, and the kernel preemption latency under various system workloads. All data are collected on a Pentium III 1-GHz machine with 128-MB RAM running RED-Linux.

A. Scheduling Delay as Performance Index

In Table 5, we show the dispatcher execution time statistics. The time measured is the time for the dispatcher to select the next job from its running queue. We have tested systems running 1, 10, and 100 jobs. The data show that the average delay is about 4 μs when the system has 100 jobs. However, the worst case delay could be as high as 96 μs.

Table 6
Kernel Latency (In Microseconds)

Workloads	min latency	average latency	max latency
No Stress	1	1	37
Memory	1	4	87
Caps − Lock	1	2	9166
Console SW	1	1	63
I/O RW	1	1	94
I/O Write	1	1	49
I/O Read	1	1	75
ProcFS	1	2	115
Fork	1	4	117
Non Real Time	14304	19880	21144

B. Preemption Latency Data

In this section, we measure the kernel latency of RED-Linux under various system loads. The workloads we have used are the same as those reported in [2]. In our test, a real-time job is run periodically with 500 μs period and 250 μs budget for a total of 100 000 periods. The job reads the time (t1), goes to sleep for 100 μs (T), then reads the time again (t2). The value $t2 - (t1 + T)$ from each period was recorded, and the min, max, and average values reported in Table 6.

We measure the system with the following workloads:

- *Memory Stress:* A large amount of memory is accessed to create successive page faults. Page faults will trigger disk accesses that may cause a long latency.
- *Caps-Lock:* In Linux, the kernel code for switching keyboard caps-lock and num-lock waits for an acknowledgment from the keyboard. This potentially could create a long latency.
- *Console-SW:* This test switches between different virtual consoles. When switching between virtual consoles, the console driver needs to copy a large video memory which may take a long latency.
- *I/O Stress:* The workload uses `read( )` and `write( )` system calls to move data between user space and kernel space. The data copying may require a long latency.
- *ProcFS:* The `/proc` file system is a pseudo file system to share data between kernel space and user space. Nonpreemptive access protection may be needed and may cause a long latency.
- *Fork Stress:* New processes are created using the `fork( )` command. Since new process needs to copy the process memory and also create new process table entry, this may cause a long delay.

Of all tests, the caps-lock workload has the largest maximum latency. This is due to the spinlock waiting for keyboard acknowledgment. The rest of them have the worst case time from 49 us to 117 us. Our test result is consistent with the data reported in [2]. The caps-lock stress test shows the importance of designing good device drivers for real-time devices. Unless we also insert preemption points in device drivers, any nonpreemptive blocking in device drivers will cause a long kernel latency.

VII. Related Work

Different from other RTOS projects (such as RT-Mach [26] and Spring Kernel [42]) that adopt one scheduling paradigm, RED-Linux implements a general scheduling framework to support various scheduling paradigms, including priority-driven, time-driven, and share-driven scheduling. Different from other real-time Linux projects (such as RT-Linux [6] and RTAI [33]), RED-Linux does not use a separate kernel (or kernel mode) to handle real-time jobs. All real-time and nonreal-time jobs are handled and scheduled by the same kernel. Real-time jobs are scheduled according to their real-time and performance constraints. Nonreal-time jobs may be scheduled with a lower priority and a fair share of system bandwidth, i.e., excluding the bandwidth used by real-time and other nonreal-time jobs.

At least two other projects have investigated multilevel real-time scheduler structure. The first was the Open System Architecture project from the University of Illinois [12]. The project was implemented under Windows NT. In their design, both levels are in the user space. The second was done by Jeffay *et al.* [19], where a system was implemented under FreeBSD. They conclude that at least one of these levels should be moved into the kernel space for performance reasons. Goyal *et al.* propose start-time fair queuing (SFQ) [14], a proportional share scheduling algorithm that works well in hierarchical environments. They implemented an architecture that uses SFQ schedulers at all points in the scheduling hierarchy. However, the timing control for each group is not enforced, only the budget is shared.

A related work for resource management in operating systems is the Linux/RK (Resource Kernel) project [31]. The main focus of Linux/RK is to provide a portable resource kernel which is independent from host operating system kernel to manage system resources. However, the portable resource kernel relies on the host OS to provide a variety of support such as an exported interface to manipulate priority levels, an exported interface for suspending and resuming execution objects, and an exported interface for acquiring events, etc. If the host OS does not have enough support, it will need to be modified. For example, in Linux/RK, Linux kernel is modified by adding callback hooks to the Linux kernel in order to catch certain scheduling points and events so that more accurate accounting information can be obtained. In comparison, the CPU dispatcher in RED-Linux takes full control of scheduling, and all these events are known in the CPU dispatcher. Linux/RK allows resource accounting to be defined by users. But job scheduling is still performed by native kernel schedulers. In other words, if the native scheduler is not designed to support real-time applications, the Linux/RK approach will not be able to enhance its capability.

VIII. Conclusion

To support real-time applications, many of today's operating systems lack the capabilities to accommodate job executions with timing constraints. They also lack the high-resolution timer and kernel latency control to reduce

the minimum response time for some time-critical devices and applications.

In the RED-Linux project, we address the latency issues of the operating system and propose several new mechanisms to allow users to control the latency. The kernel latencies are reduced by using preemption points. We also enhance the Linux scheduler by proposing a user-programmable scheduling mechanism called the general scheduling framework. We have implemented these mechanisms in RED-Linux to show its feasibility and performance. The result has proven that our work is both practical and powerful.

On the other hand, the search for an ideal real-time kernel is far from over. Although our framework has been used to support existing schedulers, new schedulers are constantly invented to support new applications, new hardware architectures, and new software capabilities. Further enhancement to our framework will be necessary to support new schedulers that use scheduling attributes not defined in our framework.

REFERENCES

[1] L. Abeni and G. C. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proc. IEEE Real-Time Systems Symp.*, Madrid, Spain, Dec. 1998, pp. 4–13.

[2] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A measurement-based analysis of the real-time performance of the Linux kernel," in *Proc. IEEE Real Time Technology and Applications Symp. (RTAS)*, Sept. 2002, pp. 133–142.

[3] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, pp. 600–625, 1996.

[4] G. Banga, P. Druschel, and J. Mogul, "Resource containers: A new facility for resource management in server systems," in *OSDI'99*, New Orleans, LA, 1999, pp. 45–58.

[5] S. Baruah, J. Gehrke, and C. G. Plaxton, "Fast scheduling of periodic tasks on multiple resources," in *Proc. 9th Int. Parallel Processing Symp.*, 1996, pp. 280–288.

[6] M. Barabanov and V. Yodaiken. Introducing real-time Linux. [Online]http://www.linuxjournal.com/article.php?sid=0232

[7] J. C. R. Bennett and H. Zhang, "WF2Q: Worst-case fair weighted fair queueing," in *Proc. IEEE INFOCOMM'96*, San Francisco, CA, 1996, pp. 120–128.

[8] L. B. David, "Scheduling support for concurrency and parallelism in the Mach operating system," *IEEE Computer*, pp. 35–43, May 1990.

[9] K. Duda and D. Cheriton, "Borrowed virtual time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler," in *Proc. ACM Symp. Operating Systems Principles*, 1999, pp. 261–276.

[10] R. Davis and A. Wellings, "Dual priority scheduling," in *Proc. IEEE Real-Time Systems Symp.*, 1995, pp. 100–109.

[11] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *J. Internetwork. Res. Experience*, pp. 3–26, 1990.

[12] Z. Deng and J. W.-S. Liu, "Scheduling real-time applications in open system environment," in *Proc. IEEE Real-Time Systems Symp.*, San Francisco, CA, 1997, pp. 308–319.

[13] S. Golestani, "A self-clocked fair queueing scheme for broadband applications," in *Proc. IEEE INFOCOMM'94*, Toronto, ON, Canada, 1994, pp. 636–646.

[14] P. Goyal, X. Guo, and H. Vin, "A hierarchical CPU scheduler for multimedia operating systems," in *Proc. Operating System Design and Implementation*, Seattle, WA, 1996, pp. 107–122.

[15] P. Goyal and H. M. Vin, "Fair airport scheduling algorithms," in *Proc. 7th Int. Workshop Network and Operating System Support for Digital Audio and Video*, St. Louis, MO, 1997, pp. 273–281.

[16] C.-C. Han, K.-J. Lin, and C.-J. Hou, "Distance-constrained scheduling and its applications to real-time systems," *IEEE Trans. Comput.*, vol. 45, pp. 814–826, July 1996.

[17] C.-W. Hsueh and K.-J. Lin, "An optimal pinwheel scheduler using the single-number reduction techniques," in *Proc. IEEE Real-Time Systems Symp.*, 1996, pp. 196–205.

[18] ——, "On-line schedulers for pinwheel tasks using the time-driven approach," in *Proc. 10th Euromicro Workshop Real-Time Systems*, Berlin, Germany, 1998, pp. 180–187.

[19] K. Jeffay *et al.*, "Proportional share scheduling of operating system service for real-time applications," in *Proc. IEEE Real-Time Systems Symp.*, Madrid, Spain, 1998, pp. 480–491.

[20] M. B. Jones and J. Regehr, "CPU reservations and time constraints: Implementation experience on windows NT," in *Proc. 3rd Windows NT Symp.*, Seattle, WA, 1999, pp. 93–102.

[21] M. B. Jones, D. Rosu, and M.-C. Rosu, "CPU reservation and time constraints: Efficient, predictable scheduling of independent activities," in *Proc. 16th ACM Symp. Operating Systems Principles*, 1997, pp. 198–211.

[22] H. Kopetz, "The time-triggered model of computation," in *Proc. IEEE Real-Time Systems Symp.*, Madrid, Spain, 1998, pp. 168–177.

[23] C. L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[24] J. W.-S. Liu, K. J. Lin, W.-K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao, "Imprecise computations," *Proc. IEEE*, vol. 82, pp. 83–94, Jan. 1994.

[25] R. Love. The Linux preemptible kernel patch. [Online]. Available: http://www.tech9.net/rml/linux/

[26] C. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *IEEE Conf. Multimedia Computing and Systems*, May 1994, pp. 90–100.

[27] M. Moir and S. Ramamurthy, "Pfair scheduling of fixed and migrating periodic tasks on multiple resources," in *Proc. 20th Annu. IEEE Real-Time Systems Symp.*, Phoenix, AZ, 1999, pp. 294–303.

[28] C. W. Mercer and H. Tokuda, "Preemptibility in real-time operating systems," in *Proc. 13th IEEE Real-Time Systems Symp.*, 1992, pp. 78–87.

[29] ——, "An evaluation of priority consistency in protocol architectures," in *Proc. IEEE 16th Conf. Local Computer Networks*, 1999, pp. 386–398.

[30] A. Morton. Linux scheduling latency. [Online]. Available: http://www.zip.com.au/akpm/linux/schedlat.html

[31] S. Oikawa and R. Rajkumar, "Portable RK: A portable resource kernel for guaranteed and enforced timing behavior," in *Proc. 5th Real-Time Technology and Applications Symp.*, Vancouver, BC, Canada, June 1999, pp. 111–120.

[32] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: The single-node case," *IEEE/ACM Trans. Networking*, vol. 1, pp. 344–357, June 1993.

[33] Real-time application interface. DIAPM, Politecnico di Milano, Italy. [Online]. Available: http://www.aero.polimi.it/projects/rtai/

[34] A. Silberschatz and P. Galvin, *Operating System Concepts*. New York: Wiley, 2001.

[35] B. Sprunt, L. Sha, and J. P. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *Real-Time Syst. J.*, vol. 1, pp. 27–60, 1989.

[36] M. Spuri and G. C. Buttazzo, "Efficient aperiodic service under the earliest deadline scheduling," in *Proc. IEEE Real-Time Systems Symp.*, Dec. 1994, pp. 2–11.

[37] B. Srinivasan *et al.*, "Firm real-time system implementation using commercial-off-the-shelf hardware and free software," in *Proc. IEEE Real-Time Technology and Application Symp.*, 1998, pp. 112–119.

[38] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and G. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *Proc. IEEE Real-Time Systems Symp.*, 1996, pp. 288–299.

[39] I. Stoica, H. Zhang, and T. S. E. Ng, "A hierarchical fair service curve algorithm for link-sharing, real-time and priority services," in *Proc. ACM SIGCOMM'97*, Cannes, France, 1997, pp. 249–262.

[40] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus, "A firm real-time system implementation using commercial off-the-shelf hardware and free software," in *Proc. IEEE Real-Time Technology and Applications Symp.*, 1998, pp. 112–120.

[41] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, pp. 1175–1185, Sept. 1990.

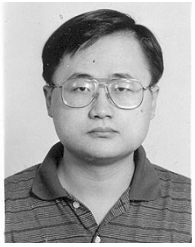
[42] J. Stankovic, D. N. Ramamritham, M. Humphrey, and G. Wallace, "The spring system: Integrated support for complex real-time systems," *Real-Time Syst. J.*, vol. 16, pp. 223–251, 1999.

[43] Y. C. Wang and K. J. Lin, "Enhancing the real-time capability of the Linux kernel," in *Proc. 5th RTCSA'98*, Hiroshima, Japan, 1998, pp. 11–20.
[44] Y. C. Wang and K.-J. Lin, "Implementing a general real-time scheduling framework in the RED-Linux real-time kernel," in *Proc. IEEE Real-Time Systems Symp.*, Phoenix, AZ, Dec. 1999, pp. 246–255.
[45] Y. C. Wang and K.-J. Lin, "The implementation of hierarchical schedulers in the RED-Linux scheduling framework," in *Proc. 12th Euromicro Workshop Real-Time Systems*, Stockholm, Sweden, June 2000, pp. 231–238.
[46] S. Wang, K. J. Lin, and Y. C. Wang, "Hierarchical budget management in the RED-Linux scheduling framework," in *Proc. 14th Euromicro Workshop Real-Time Systems*, Vienna, Austria, June 2002, pp. 76–83.
[47] S. Wang, Y. C. Wang, and K. J. Lin, "Integrating priority with share in the priority-based weighted fair queuing scheduler for real-time networks," *Real-Time Syst. J.*, vol. 22, pp. 119–149, 2002.

Kwei-Jay Lin (Senior Member, IEEE) received the B.S. degree in electrical engineering from the National Taiwan University, Taipei, Taiwan, R.O.C., in 1976, and the M.S. and Ph.D. degrees in computer science from the University of Maryland, College Park, in 1980 and 1985, respectively.

He is a Professor in the Department of Electrical Engineering and Computer Science at the University of California, Irvine (UCI). He was an Associate Professor at the University of Illinois at Urbana-Champaign before he joined UCI. His research interests include real-time systems, scheduling theory, operating systems, and web-based systems. He has published more than 100 papers in academic journals and conference proceedings.

Dr. Lin is an executive committee member of the IEEE Technical Committee on Real-Time Systems. He is a co-chair of the IEEE Task Force on E-Commerce. He is an Associate Editor of IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS. He was an Associate Editor of the IEEE TRANSACTIONS ON COMPUTERS from 1996 to 2000. He has served on program committees of many international conferences and workshops. His most recent conference duties include the General Chair of 2003 IEEE Conference on E-Commerce, Workshop Chair of EEE'03 in Taipei, and Program Vice Chair for ICDCS 2004.



Yu-Chung Wang received the B.S. degree in physics from the National Cheng-Kung University, Tainan, Taiwan, R.O.C., the M.S. degree in physics from the National Taiwan University, Taipei, Taiwan, and the Ph.D. degree in computer engineering from the University of California, Irvine, in 2001.

He is currently the Chief Technology Officer of a Linux startup, REDSonic Inc., Santa Ana, CA. His research interests include operating systems, embedded systems, real-time scheduling, and system software and middleware.