Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior

Shuichi Oikawa and Ragunathan Rajkumar

Real-time and Multimedia Systems Laboratory

School of Computer Science

Carnegie Mellon University

5000 Forbes Avenue, Pittsburgh, PA 15213 USA

E-mail: {shui+,raj+}@cs.cmu.edu

Abstract

Portable RK is a portable implementation of a *resource kernel*, a resource-centric approach to build a real-time kernel that provides explicit timely, guaranteed and enforced access by applications to system resources [13]. Portable RK is designed to work with widely available operating systems with minimal changes. This facilitates experimentation in a familiar software environment and helps the faster deployment of research results. Execution in resource kernels is directly based on OS-enforced resource reservation [7]. As a result, an application can request the reservation of a certain amount of a resource, and the kernel can guarantee that the requested amount is exclusively available to that application. In this paper, we describe the design and implementation of Portable RK called *Linux/RK* that resides within the Linux operating system. The evaluation results show that Portable RK in the form of Linux/RK gives direct control over timely resource utilization by applications and that its overhead costs are small enough to be negligible.

1 Introduction

A *resource kernel* [10, 12] is a resource-centric approach for building real-time kernels that provide timely, guaranteed and enforced access to system resources. In this paper, we present the design and implementation of a portable version of a resource kernel called *Portable RK*[1]. The objective of Portable RK is such that a common code base can be used in conjunction with different operating systems including real-time kernels as well as commodity (non-real-time) operating systems. The discussion in this paper centers on the use of Portable RK within the Linux operating system, a combination that we refer to as *Linux/RK*. A main function of an operating system kernel is to multiplex available system resources across multiple requests from several applications. For example, non-real-time operating systems allocate a time-multiplexed resource to an application based on fairness metrics measured over discrete intervals of time. The key philosophy behind the resource kernel is that precise timing guarantees and temporal protection between applications can be obtained by imposing a well-defined resource usage model on time-multiplexed resources. In other words, an application running on a resource kernel can request the reservation of a certain amount of a resource, and the kernel can guarantee that the requested amount is exclusively available to the application. Since continual monitoring of resource usage is carried out by the resource kernel to enforce resource usage by any application, such a guarantee of resource allocation gives an application the knowledge of the amount of its currently available resources. A QoS manager or an application itself can then optimize the system behavior by computing the best QoS obtained from the available resources [13, 17].

The resource kernel concept has been tightly integrated into the RT-Mach operating system[2], and allows guaranteed, timely and enforced access by applications to CPU cycles [13], disk bandwidth [9] and (lately) network bandwidth. However, since the RK concept itself can be independent of other OS aspects, from a practical point of view, it is

[1] RK stands for Resource Kernel.
[2] Please see "$http://www.cs.cmu.edu/{\sim}rtmach$".

useful to consider how RK mechanisms, policies and interfaces can be made OS-independent. There are two goals for portability:

- *A portable resource kernel interface:* Having a common RK interface across platforms is aimed at ease of programming on different platforms. Given the Portable RK API as a substrate for a resource management framework, programmers benefit from this common API and resource management abstractions. When programmers are writing a program for two different operating systems, they only need to deal with the differences in the OS APIs.

- *A portable implementation code base:* Explicit dependencies of the resource kernel implementation on a particular kernel architecture (like the RT-Mach implementation of the resource kernel [13]) will force the re-writing of RK functionality for other operating systems. It would therefore be ideal if Portable RK can be designed not to depend on a specific kernel architecture. In essence, Portable RK must include in an OS-independent way that the common components needed to implement RK abstractions. It must be noted that there must exist a means to interface Portable RK with its "host" operating system.

In this paper, we describe the design and implementation of our Portable RK that operates within the Linux operating system. We refer to this integrated system as *Linux/RK*. Linux/RK is the Linux kernel [4] that includes Portable RK as a core (but distinct) subsystem. We chose the Linux kernel as a basis of the first portable implementation because of its popularity and availability on a wide variety of platforms. Since Linux source code is freely available, we are able to investigate the support necessary from commodity operating systems with and without source code availability. While implementing Portable RK within the Linux environment, we have explicitly intended to keep the number of modifications to the original Linux kernel as small as possible. Thus, we have been developing Portable RK as an independent module from the Linux kernel, and only a small number of *callback hooks* were introduced into the Linux kernel to combine it with Portable RK.

Portable RK currently supports only processor reservation (and not disk bandwidth reservation or network bandwidth reservation). Nevertheless, our evaluation results included in this paper are encouraging and stimulate the need for additional support. We will summarize ongoing work in Section 6.

1.1 Related Work

RT-Mach has been evolving from the original implementation [18], and now includes resource kernel support [12] that facilitates reservations on multiple resource types. *CPU reserves* [7] were first introduced in RT-Mach, and network and disk bandwidth reservations are also available [3, 9]. Resource kernel functionality in RT-Mach RK is, however, tightly coupled with the microkernel. Portable RK is, by contrast, an independent kernel subsystem. It is modular and portable, so that it can be easily interfaced to commodity operating system kernels thereby making them resource kernels as well.

KURT [16] and Real-Time Linux [1] are real-time extensions to the Linux kernel with each taking a different approach. Real-Time Linux is a combination of a simple real-time executive and patches to the Linux kernel to make it run on the executive. Real-time tasks run on the executive and can communicate with processes of Linux through a device interface. KURT provides firm real-time functionality, which is defined as a hard timing requirement and a soft deadline. A normal real-time task in KURT resides in the Linux kernel as a form of a loadable kernel module and can utilize Linux functions. KURT features a high-resolution timer as in RT-Mach RK to satisfy hard timing requirements. Neither of these Linux real-time extensions provides resource kernel features, such as guaranteed reservation and enforcement.

There are several commercially available real-time extensions to Microsoft Windows NT [8]. Their approaches are similar to RT-Linux. Typically, a real-time executive is buried at the bottom of the NT kernel and provides real-time functionality with real-time tasks. Thus, real-time tasks cannot utilize NT's services. DREAMS [15] is a user-level real-time extension to Microsoft Windows NT. It took a similar approach to ours for controlling applications' resource utilization, but its capability is very much limited because of its user-level implementation. It must also be noted that the EPIQ project at the University of Illinois at Urbana Champaign, the HARTIK kernel from Italy and the Nemesis OS at Cambridge have similar goals as the resource kernel work. The EPIQ project focuses on Windows NT with source code access, the HARTIK kernel emphasizes variants of earliest deadline scheduling in admission control and scheduling, while the Nemesis OS focuses on building a clean implementation from scratch minimizing interference arising from between application interactions as much as possible.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 describes the abstractions provided by Portable RK and how they constitute Portable RK. Section 3 describes the implementation of Portable RK and Linux/RK. Section 4 shows the results of measurements and evaluates Linux/RK. Section 5 discusses the portability of the implementation without access to the source code. Finally, Section 6 summarizes the paper.

2 Resource Management Architecture

This section describes the abstractions provided by Portable RK and how they constitute Portable RK. There are two basic abstractions in Portable RK, a *reserve* and a *resource set*. A reserve represents a well-defined share of a single system resource, such as CPU cycles, disk bandwidth, network bandwidth, and physical memory [7]. Typically, a resource scheduler exists for each resource, which actively manages its underlying resource to distribute the shares of the resource to allocated reserves. A resource set is a grouping of reserves on different resources into a single set that is accessible by user processes. In other words, a resource set is the abstraction by which user processes get access to system resources. When a user process bounds itself to a resource set, the process obtains a guaranteed execution

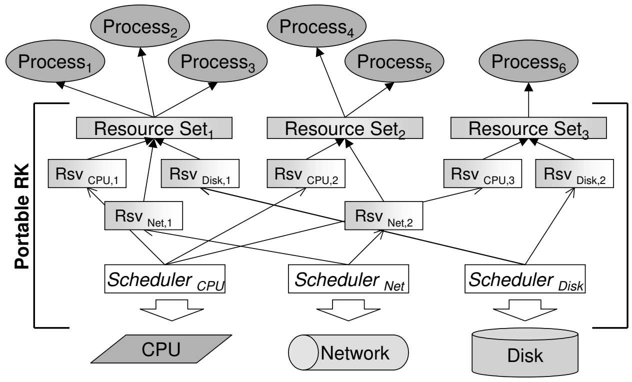

Figure 1: Architecture of components constituting Portable RK

environment corresponding to the reserves within the resource set.

Figure 1 depicts the overall architecture of Portable RK. The bottom-most layer consists of the physical system resources. The next higher layer consists of the Portable RK abstractions which consist of one scheduler associated with each system resource, and resource sets, each of which contains a reserve on one or more system resources. Application processes access system resources in a timely and guaranteed fashion by creating and executing within the context of resource sets. The following sections describe the details of reserves, their support mechanisms, resource sets, and the API exported to user processes.

2.1 Reserves

A *reserve* represents a share of a single computing resource. Such a resource can be CPU time, physical memory pages, a network bandwidth, or a disk bandwidth. A certain amount of a resource share is reserved for use by the programs. A reserve is implemented as a kernel entity; thus, it cannot be counterfeited. The kernel keeps track of the use of a reserve and will *enforce* its utilization, when necessary. Appropriate scheduling and enforcement of a reserve by the resource kernel guarantees that the reserved amount is always allocated for it.

A reserve can be time-multiplexed or dedicated. Temporal resources, such as CPU cycles, network bandwidth and disk bandwidth, are time-multiplexed, and spatial resources like memory pages are dedicated. A time-multiplexed resource has the following primary reserve parameters[3]: $\{C, D, T\}$, where T represents a recurrence period, C represents the processing time required within T, and D is the deadline within which the C units of processing time must be available within T.

2.1.1 Reserves: Abstract and Real

Reserves contain certain amounts of resource shares and control their utilization. A reserve may represent one of many different types of resources such as CPU cycles and network bandwidth. Different types of resources have their own

[3]Please see [13] for other “secondary” parameters.

create	creates a reserve. Specifying a list of resource reservation parameters, both abstract and real reserve objects are created.
destroy	destroys a reserve. Both abstract and real reserve objects are destroyed.
start_account	starts the accounting on a reserve.
stop_account	stops the accounting on a reserve.
replenish	replenishes a reserve.
enforce	enforces a reserve.

Table 1: Interface provided by abstract reserves

accounting information and their own ways to deal with resource management. At the same time, reserves need to provide a uniform interface, otherwise modifications are required each time a new resource type is added. Therefore, a reserve is decoupled into *abstract* and *real reserves*. An abstract reserve implements the functionality common across all reserves and provides a uniform interface. A real reserve implements resource-type-specific portions and exports functions that adhere to the uniform resource management interface. Abstract and real reserves are always paired. When a reserve is created, each of them is created and is coupled with each other. The distinction is useful because it requires only that real reserve be implemented for a new resource type. Table 1 shows the standard interface provided by abstract reserves.

2.1.2 Guarantee Mechanisms

Real reserves implement the following mechanisms that guarantee resource utilization based on reservation.

- *Admission control*: Portable RK performs an admission control test on a new request to determine if it can be accepted or not. If the request can be admitted, a reserve based on the requested parameters is created.
- *Scheduling policy*: A scheduling policy controls dynamic resource allocation, so that an application can receive its reserved amount of a resource share.
- *Enforcement*: Portable RK enforces the use of a resource by an application based on its allocated reserves. An enforcement mechanism prevents a resource from being used more than its reserved amount.
- *Accounting*: Portable RK tracks how much of a resource share that an application has already used. The scheduling policy and the enforcement mechanism use this information for their decision making. An application, a QoS manager or a real-time visualization tool can also query this information for observation and dynamic resource allocation control purposes.

2.1.3 Reservation Types

When a reserve uses up its allocated time units C within an interval T, it is said to be *depleted*. A reserve that is not depleted is said to be an *undepleted* reserve. At the end of the current interval T, the reserve will obtain a new quota and is said to be *replenished*. Consider a process (or thread) that is bound to a reserve. During the execution of

	Enforcement	Replenishment
Hard	The reserve will not be scheduled on depletion until it is replenished.	Any excessively used time in the current period is subtracted from the reserved time in the next period, so that the actually consumed time does not exceed the total reserved time.
Farm	The reserve will be scheduled for execution on depletion only if no other undepleted reserve or unreserved resource use can be scheduled.	The share of the next period is adjusted from the average CPU utilization so that the actually consumed time becomes close to the total reserved time (over long intervals of time).
Soft	The reserve can be scheduled for execution at background priority on depletion along with other unreserved resource use and depleted reservations.	The reserved time C is always replenished for the next period.

Table 2: Reserve modes in the extended resource management model

the process, the reserve gets depleted but the process remains ready to execute. In our original resource management model [13], three forms of reserves (called hard, firm and soft reserves) determined how processes bound to a depleted reserve behave until the depleted reserve got replenished. Processes bound to a depleted hard reserve, for example, could not execute until the reserve got replenished. In contrast, processes bound to a soft reserve could execute at background priority (less than real-time processes) until the reserve got replenished. In summary, the reserve type determines the degree of strictness in the enforcement of *excess* resource consumption on a reserve.

In addition to the above hard, firm and soft enforcement on a reserve, an added requirement is addressed in the context of the Portable RK. When RK is implemented in monolithic or non-preemptive kernels, it is possible that a process is executing (say) a system call within the kernel when its corresponding reserve expires. However, since the process cannot be suspended inside the kernel, it would be allowed to complete its execution to the point of return from kernel mode to user space. As a result, this reserve can consume more than its allocated C for this particular period of the reserve.[4] The question that we ask is whether this excess consumption is unaccounted for in the corresponding reserve. An alternative is that this excess consumption can be charged to the reserve at its next replenishment.

To enable such more precise control of a reserve, Portable RK supports an *extended resource management model*, in which the behavior of a reserve can be specified independently for each action of enforcement and replenishment. In the Portable RK, either enforcement or replenishment can now take one of hard, firm, and soft as its value, and they can be specified independent of one another. For example, a reserve can be created with a specified behavior of hard enforcement and soft replenishment. We expect various application environments to have custom default values on reserve types based on their most common occurrences. Table 2 summarizes the specific behavior of a reserve type based on values assigned to their enforcement and replenishment actions.

[4] Admission control in a resource kernel should take into account this effect when reserves are created.

2.2 Resource Sets

A *resource set* represents a set of reserves. it *groups* necessary reserves for the job of user applications. One or more programs are attached to a resource set, which provides the exclusive use of its reserved amount of resource shares with those programs. Such grouping of resources makes it easier to examine and compare the utilization of each resource in a resource set. If the kernel or a QoS manager finds an imbalance in resource utilization, an application can be notified and will be able to change its QoS parameters dynamically if needed to balance the utilization.

From the user program's point of view, a resource set is the interface to reserves to which user programs have access. A user program creates necessary reserves and attaches them to its resource set. A user program can be bound to at most a single resource set. Even when an execution object uses only a single reserve, it has to create a resource set and attach its only reserve to the resource set.

Although the notion of resource sets remains the same in Portable RK, the representation of a running user program can vary from OS to OS. A "process" is an execution object of a user program in the Linux kernel, and Portable RK in Linux/RK holds its reference in a resource set. While there is a single thread of control in a process for the Linux kernel and many UNIX variants, there can be multiple threads in a single task for Mach, Windows NT and other modern operating system kernels. It is possible to attach threads in the same task (process) to different resource sets in such a system. Since a resource set groups reserves to accomplish a job rather than just to run a program, it is more natural to bind a thread and a resource set than to bind a task and a resource set.

For implementation convenience, each process contains a reference to its resource set, and each resource set holds the references to its attached reserves and processes using the resource set. In addition, a resource set also greatly simplifies the mechanism by which the current active reserves on various resources can be determined at run-time.

2.3 API

Portable RK provides an API (Application Program Interface) for use by user programs. They use this API to create reserves and resource sets, and then to attach their processes to the created resource sets. Table 3 summaries the basic API of Portable RK.

3 Implementation

This section describes the implementation of Portable RK and Linux/RK. First, the implementation details of CPU reserves are described. Then, Callback hooks, which were inserted in the Linux kernel to interface it with Portable RK, are described. The support of a timestamp counter and a high resolution timer is described next. Such support is more important in resource kernels than in traditional kernels, since accurate time management is a key requirement for timeliness enforcement in resource kernels. Finally, the use of the Proc filesystem is described. The support of the Proc file system makes it easier for applications to obtain information on the current statuses of kernel internals.

rk_resource_set_create	creates an empty resource set. It is an empty since no reserve is attached to it. An application program, which uses Portable RK, begins with creating a resource set. Then, the created resource set is specified to create reserves, which are attached it.
rk_resource_set_destroy	destroys a resource set. When a resource set is destroyed, the attached reserves are also destroyed at the same time. An application program destroys a resource set, which is no longer necessary, before it exits.
rk_resource_set_attach_process	attaches a process to a resource set. Even when an application creates a resource set, it has to attach itself to the created resource set in order to use the reserves in the resource set; thus, such an attachment comes after the creation of reserves. Multiple processes can be attached to a single resource set.
rk_resource_set_detach_process	detaches a process from a resource set. If a resource set is not created by an application but another process (possibly a QoS manager) provides the resource set with it, it has to detach itself from the resource set.
cpu_reserve_create	creates a CPU reserve. Specifying a list of resource reservation parameters, both abstract and real reserve objects are created.

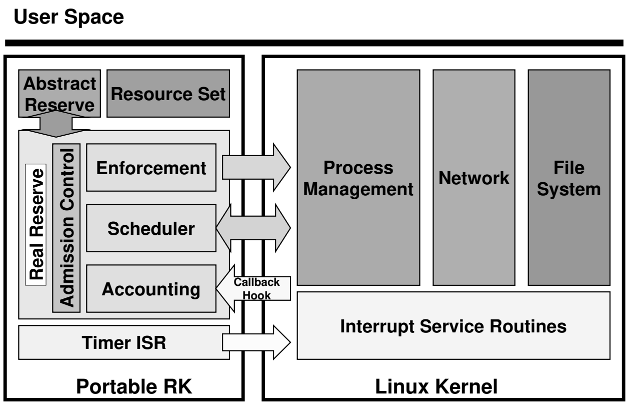

Figure 2: Linux/RK Architecture composed of Portable RK and Linux Kernel

Figure 2 depicts the overall architecture of Linux/RK, which describes how Portable RK and the Linux kernel interact with each other.

3.1 CPU reserves

The implementation of CPU reserves includes the mechanisms for admission control, scheduling, enforcement, replenishment, and accounting to guarantee resource utilization based on reservation. Admission control and scheduling mechanisms are decoupled from their policies, which depend on a target resource type. For example, the admission control scheme for disk bandwidth management [9] is very different from that for CPU cycle allocation [13]. Considering the requirements of the applications running on a target system, a scheduling policy must be chosen, and the scheduling policy determines its admission control policy. The admission control is implemented solely in Portable RK, and is independent from the host operating system kernel, which is the Linux kernel in Linux/RK. On the other hand, scheduling, enforcement, replenishment, and accounting mechanisms depend on the host operating system kernel. Since the host operating system kernel manages execution objects, which are processes in case of Linux/RK, Portable RK needs to query the host operating system kernel about the status of execution objects. When Portable RK needs to manipulate them, it must follow the way of the host operating system kernel to manage them possibly by using the exported functions. Otherwise, it is difficult to keep the consistency of execution objects; thus, their portable mechanism must be devised for Portable RK. The details of scheduling, enforcement, replenishment, and accounting mechanisms are described as follows.

3.1.1 Scheduling

Most modern operating system kernels provide a fixed-priority preemptive scheduler. Portable RK uses it to emulate various CPU scheduling policies, which guarantee the shares of resources along with their admission control policies. These scheduling policies put execution objects in certain order based on the timing attributes of execution objects. Following the order defined by those scheduling policies, Portable RK assigns priority levels of fixed-priority scheduling policy to the execution objects. The following two issues must be addressed for the correct scheduling of execution objects within a resource kernel:

- *Assignment of priority levels*: The assignment of priority levels is not always fixed. Some scheduling policies define their ordering in a static way. In other words, the ordering is always the same at any given time. An example is RM (Rate-Monotonic) scheduling policy, which defines the ordering based on the periods of execution objects. The assignment of priority levels is changed for these scheduling policies only when a new reserve is created and it is attached to an execution object. The other policies define their ordering in a dynamic way. In other words, the ordering may change as time passes. An example is EDF (Earliest Deadline First) scheduling policy, which defines the ordering based on how close execution deadlines are. The assignment of priority levels is changed for these scheduling policies as time passes, and a support mechanism is necessary for the scheduler to respond to timing events.
- *Execution Suspension and Resumption*: As reserves get replenished and depleted, execution objects bound to them must be resumed and suspended respectively (or priorities raised and lowered respectively depending upon the reserve type). A mechanism is therefore necessary to schedule execution objects bound to a reserve when its turn comes to execute, and to de-schedule execution objects when its turn expires.

Portable RK introduces a three-tier scheduler architecture to address the above issues. Two Portable RK scheduler threads[5] run at the highest and lowest priority levels in the priority range of the scheduler respectively.[6] The highest priority scheduler thread performs the enforcement and replenishment of reserves and the priority adjustment of execution objects. It wakes up for performing the scheduling jobs, and sleeps until the next event. It runs at the highest

[5]These are threads of control in the kernel. Execution objects at the user level in the Linux kernel are processes, yet it supports kernel threads for efficiently implementing some services. Other UNIX variants also support in-kernel threads in different ways.

[6]Priority levels between the highest and lowest levels are dynamically assigned to processes attached to reserves.

priority level, so that it can preempt all other threads and run to perform its scheduling jobs. The lowest priority scheduler thread takes care of execution objects that do not fit into the available range of priority levels. Portable RK keeps the list of those execution objects. When the lowest priority scheduler thread is invoked, there is no runnable execution objects at higher priority levels. The lowest priority scheduler thread chooses the next thread to run and assigns it to the next higher priority level, which is reserved for this purpose. When the execution of this execution object is finished, the lowest priority scheduler thread is invoked again and continues the loop of choosing the next thread to run.

In Linux/RK, we have opted for a performance optimization and replaced the highest priority scheduler thread by a timer ISR[7].

3.1.2 Enforcement and Replenishment

The enforcement and replenishment of reserves include adjusting the priority levels of execution object, and suspending and resuming execution objects. Those operations depend on the host operating system kernel, and they are sometimes not directly supported via a well-defined API.

In Linux/RK, wait queues provided by the Linux kernel are used to suspend processes and to resume them later. The Linux kernel uses wait queues internally to block processes until certain conditions are met. A pair of functions, `sleep_on()` and `wake_up()`, is used to suspend a process by putting it in a wait queue and to resume processes in a wait queue by removing them from it respectively. While it is possible to suspend and to resume a process by sending SIGSUSP and SIGCONT signals, these are insecure since a suspended process can be resumed by sending SIGCONT signal.

3.1.3 Accounting

Accounting on reserves requires acquiring events of context switching between execution objects. When an execution object with a CPU reserve starts running, Portable RK starts its accounting. When it yields to another execution object, its accounting stops. It can yield to another execution object at various points inside the kernel because of many reasons. The scheduler is eventually called to choose the next execution object to run; thus, catching scheduling events is the most appropriate and convenient way for the accounting on reserves.

In Linux/RK, callback hooks are introduced into the Linux kernel for this purpose. Callback hooks send appropriate scheduling events to Portable RK, which uses that information to perform accounting on reserves. The details of callback hooks in Linux/RK are described next. Some commodity operating system kernels provide similar features in order to trace the status of execution objects.

[7] Interrupt Service Routine.

3.2 Callback Hooks

As mentioned already, in Linux/RK, Portable RK resides within the Linux kernel. While the Linux kernel provides normal operating system functions, Portable RK augments those functions to make it a resource kernel. Portable RK accomplishes this by implementing reserves and resource sets, along with the necessary modules for admission control, resource scheduling, resource usage accounting, and enforcement. To make the Linux kernel and Portable RK work cooperatively, there must be a means to connect and interact with each other.

Callback hooks interface the Linux kernel to Portable RK in order to introduce resource kernel abstractions into the Linux kernel. Those hooks catch relevant scheduling points in the Linux kernel, and send these events to Portable RK. Portable RK re-uses the built-in functions within the Linux kernel to control kernel entities, such as processes and device drivers.

Currently, the following callback hooks have been implemented in the Linux kernel:

- schedule_callback hook: is within schedule(), the scheduling function of the Linux kernel.
- interrupt_callback hook: intercepts Linux interrupt handling.
- interrupt_out_callback hook: is used to notify Portable RK when the processing of an interrupt is finished.
- kernel_out callback hook: is used to notify Portable RK when the execution is going back from kernel mode to the user level.

3.3 Accurate Time Management

In a resource kernel, high-precision clocks and timers are essential to maintaining the fidelity of its abstractions. The combination of a timestamp counter and a high-resolution timer contributes to improving the precision of resource management. A high-resolution timestamp counter, built into most modern CPUs, provides the basis for maintaining a high-resolution clock for use by Portable RK. The representation of time in Portable RK for use in accounting and scheduling is based on the values from this timestamp counter. A high-resolution timer [12, 16] supports precise enforcement of reserves and is implemented by using the one-shot mode of the ISA clock timer chip in PC compatible systems.

In Linux/RK, the interrupt_callback hook calls the ISR (Interrupt Service Routine) in Portable RK, and processes timer interrupts. Interrupts are propagated to the Linux kernel as needed to ensure Linux compatibility.

3.4 Exporting Reserve Status Information to User Space

The Portable RK contains information about the current status of each reserve and execution objects bound to each reserve. This dynamically changing information, if available in user space, can be accessed in real-time by QoS managers monitoring application usage of system resources or by visualization tools that display the real-time usage of resources by applications.

Resource set creation	13.78 μsec
CPU reserve creation	10.94 μsec
Process attachment	5.72 μsec
Resource set destroy	20.76 μsec

Table 4: Costs to set up and to destroy a resource set

The export of reserve status to user space in Linux/RK is accomplished through the *Proc* filesystem supported by Linux. It provides, in a portable way, information on the current status of the Linux kernel and running processes. Similarly, Linux/RK uses the Proc filesystem for providing information on the hardware platform, the reservation status, and the status of resource sets, and reserves. Hardware platform information includes CPU type and the number of cycles executed per second. Reservation status information shows that how much share of a resource is currently reserved. Information on a resource set includes the reserves attached to it. Information on reserves includes the current, minimum, and maximum utilization of their underlying resources.

4 Evaluation

We evaluated Linux/RK by performing several benchmarks. All benchmarks were performed on a 200MHz Pentium Pro PC with 256KB L2 cache and 64MB of RAM, and the times were measured using the built-in high-resolution timestamp counter. As mentioned before, Portable RK currently supports only CPU reserves and our performance evaluation therefore only includes the results of how stable and predictable is CPU consumption to reserved CPU applications. The support for and evaluation of network bandwidth reserves is actively ongoing and will be presented in a future report.

4.1 Basic Costs

This section describes the basic costs of Linux/RK. First, the costs to set up a process with a resource set and a CPU reserve are described. Then, the run time costs of using them are described.

4.1.1 Set-Up Costs

Table 4 shows the average costs to create and destroy a resource set and a CPU reserve. The total set-up cost is 30.44 μsec including the creation of a resource set and a CPU reserve and the attachment of a process to the created resource set. The cost to destroy a resource set includes the cost to destroy the CPU reserve attached to the resource set. The total cost to set up and destroy a resource set and a CPU reserve is 51.20 μsec. Combined with the cost to create and destroy a process,[8] the cost added by Portable RK is 14%. Thus, Portable RK adds only moderate costs to the native

[8]The combined cost of `fork()` and `exit()` is 314.06 μsec, which was measured by using the *lmbench* benchmark suite [6].

Start accounting	0.56 μsec
Start accounting (program timer)	6.08 μsec
Stop accounting	0.39 μsec
Replenishment	0.97 μsec
Replenishment (resume process)	1.68 μsec
Enforcement (suspend process)	0.08 μsec

Table 5: Run-time costs of a CPU reserve

cost of starting and ending processes.

4.1.2 Run-Time Costs

Table 5 shows the average run-time costs of managing a CPU reserve. The cost to start accounting includes the costs to calculate the CPU time left within the current period and to set up its enforcement timer. Setting up an enforcement timer includes programming the ISA clock timer only when the timer expires before the next Linux timer interrupts. The cost to stop accounting includes the costs to calculate the CPU time used from the start time and to sum up the used time to the total used time. The sum of the costs to start and stop accounting are 6.47 μsec and 0.94 μsec with and without programming the high resolution timer, respectively. The cost of replenishment includes the cost to calculate the statistics information. The cost to resume a process is included when necessary. The cost of enforcement is the cost of suspending a process. Their sum is 1.76 μsec, which is the cost to enforce and replenish a CPU reserve including suspending and resuming a process

If a process starts running and then is suspended by enforcing its reserve, Portable RK adds the cost of at most 8.23 μsec for a single process per period T without yielding. If a process yields its execution, the cost per period is increased by the number of the times a process yields its execution.

The costs of accounting described above heavily rely on the costs to program the ISA clock timer. The ISA clock timer on PC compatibles unfortunately resides on the slow ISA bus, and its programming requires setting two values to its I/O port.[9] Portable RK uses 64-bit integer numbers to represent times to achieve higher precision, but compared to timer programming costs, the costs to read the timestamp counter and those of 64-bit arithmetic are negligible. The cost to read the timestamp counter is 0.20 μsec. To see how efficient this cost is, the least expensive system call in Linux, `getpid()`, takes 1.79 μsec, almost an order of magnitude higher. Even if a program executes only for 1 msec, the cost of its accounting is less than 1% of its computing time. We conclude that this cost is small enough for programs that perform meaningful jobs to be able to ignore. Finally, it is useful to note that these overhead costs will decrease even farther as processors become faster.

[9] One more dummy value is set to an unused I/O port to avoid making the ISA clock timer chip disordered.

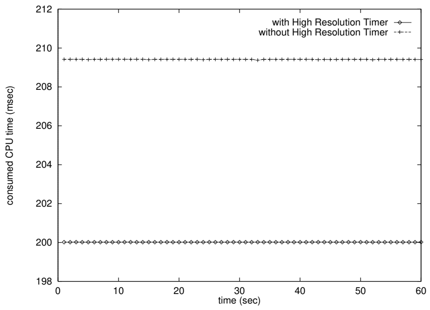

Figure 3: Enforcement of CPU Time Utilization with/without High Resolution Timer Support

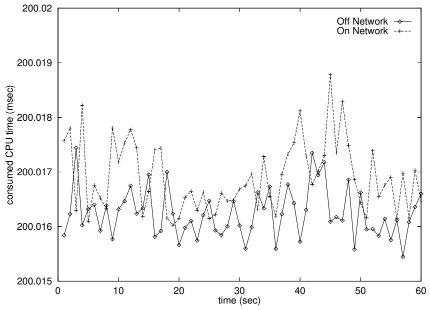

Figure 4: Enforcement of CPU Time Utilization with Network Up/Down

4.2 Accuracy of Enforcement

This section evaluates the accuracy of enforcement in Linux/RK. First, the contribution of high resolution timer support to the accuracy is described. Then, the effect of network activity, which causes non-preemptive interrupt processing, is described.

4.2.1 High Resolution Timer Support

To evaluate the effectiveness of high resolution timer support, a benchmark program ran with a processor reserve which reserved 200 msec of CPU time every 1 second. Figure 3 shows the CPU cycles actually consumed with and without the support of a high-resolution timer. Without high-resolution timer support, the program always overran for very close to 10 msec. This is directly due to the fact that Linux updates its internal time-clock at increments of 10 msec. On the other hand, with high-resolution timer support, the consumed CPU cycles are almost always 200 msec during every 1 second interval.

4.2.2 Effect of Network Activity

Figure 4 shows the effect of peripheral devices on the precision obtained with the high-resolution timer. The benchmark program ran again, when network I/F devices were up and down, with a reserve of 200 msec CPU time every 1 sec. The system was being connected to the busy departmental LAN at the CS Department at Carnegie Mellon when network

CPU Reserve	CPU Time (msec)	Period (msec)
1	10	500
2	8	400
3	6	300
4	4	200
5	2	100
6	1	50
7	0.6	30
8	0.4	20

Table 6: Task Set used for Enforcement of CPU Time Utilization with Two, Four, and Eight Processes

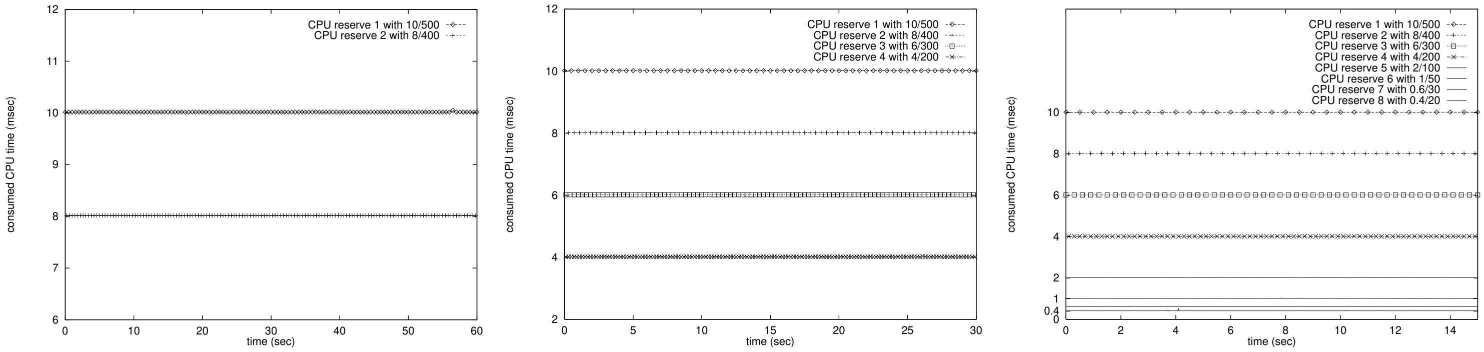

Figure 5: Enforcement of CPU Time Utilization with Two, Four, and Eight Processes

I/F devices are up. The differences are negligible although having no network connection contributes to slightly better results.

4.2.3 Multiple Processes

Figure 5 shows the CPU cycles actually consumed when two, four, and eight processes in infinite loops ran simultaneously. Table 6 shows the parameters of reserves (with hard enforcement) attached to 'infinite-loop' processes. When two, four, and eight processes ran, CPU Reserve 1 and 2, 1 to 4, 1 to 8 were used, respectively. The results show that Portable RK can rigidly enforce their CPU times even when CPU reserves with different CPU times and periods are run concurrently. When a CPU reserve with very short reserved CPU time and a short period is created, its process can still obtain the precise amount of CPU time and it does not affect the other processes running concurrently.

4.3 More Control Facilities

4.3.1 QoS Control

Figure 6 shows the CPU cycles actually consumed when one, two, and four infinite-loop process(es) ran simultaneously and changed their shares of CPU time dynamically while running. Table 7 shows the parameters of *hard* reserves attached to processes. When one, two, and four process(es) ran, CPU Reserve 1, 1 and 2, 1 to 4 were used, respectively.

CPU Reserve	Period (msec)	CPU Time (msec)					
		0 sec	10 sec	20 sec	30 sec	40 sec	50 sec
1	500	75		100		50	
2	250	30		20		40	
3	200	10	20		30		40
4	100	25	20		15		10

Table 7: Task Set used for Enforcement of CPU Time Utilization with QoS Control

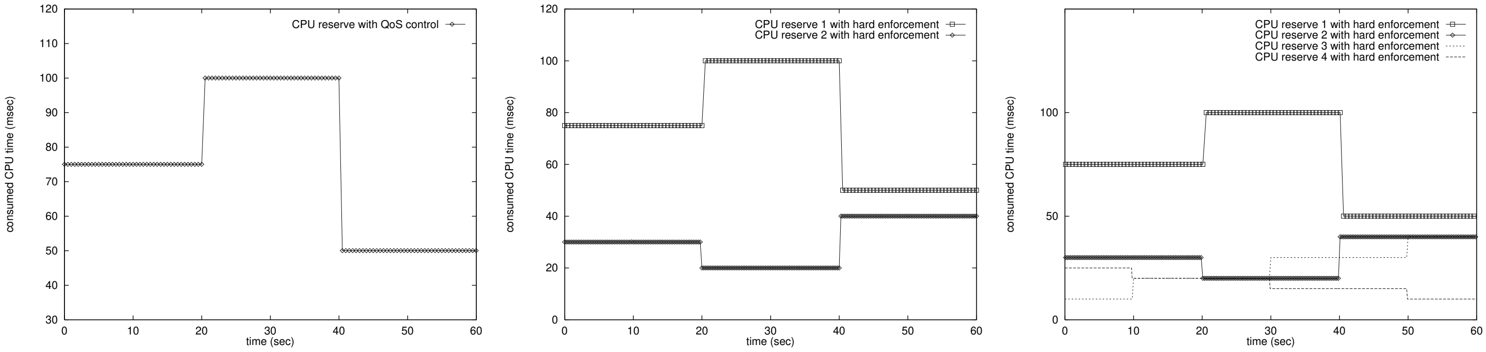

Figure 6: Enforcement of CPU Time Utilization with QoS Control

Their CPU times were changed at the indicated times. The results show that the CPU times obtained by processes change promptly when the parameters of their reserves are changed. Again, these changes do not affect the behavior of the other processes with reserves of hard enforcement and replenishment).

4.3.2 Reserves of Hard and Soft Enforcement

Figure 7 shows the CPU cycles actually consumed when two, three, and four processes with reserves of hard or soft enforcement ran simultaneously while processes with reserves of hard enforcement were changing their shares of CPU time. Table 8 shows the parameters of reserves attached to processes. When two, three, and four processes ran, CPU Reserve 1 and 2, 1 to 3, 1 to 4 were used, respectively. Their CPU times were changed at the indicated times. The results show that processes with reserves of hard enforcement can stably obtain their reserved shares of CPU times while processes with reserves of both hard and soft enforcement run simultaneously. Note that these processes do *not* exceed their specified reserved amounts of CPU time.

In contrast, infinite-loop processes bound to reserves with soft enforcement obtain *at least* the reserved amount of CPU time and actually obtain more whenever possible.

4.3.3 Reserves of Hard and Soft Replenishment

Figure 8 shows the CPU cycles actually consumed when two and three infinite-loop processes with reserves of hard, soft or firm replenishment *and* hard enforcement ran simultaneously. Table 9 shows the parameters of reserves attached to processes. When two processes ran, CPU reserves 1 and 2 were used, and when three processes ran, reserves 1

CPU Reserve	Enforcement	Period (msec)	CPU Time (msec)					
			0 sec	10 sec	20 sec	30 sec	40 sec	50 sec
1	Hard	1000	200		300		100	
2	Soft	1000	100					
3	Soft	500	50					
4	Hard	500	50	75		100		125

Table 8: Task Set used for Enforcement of CPU Time Utilization with Reserves of Hard and Soft Enforcement

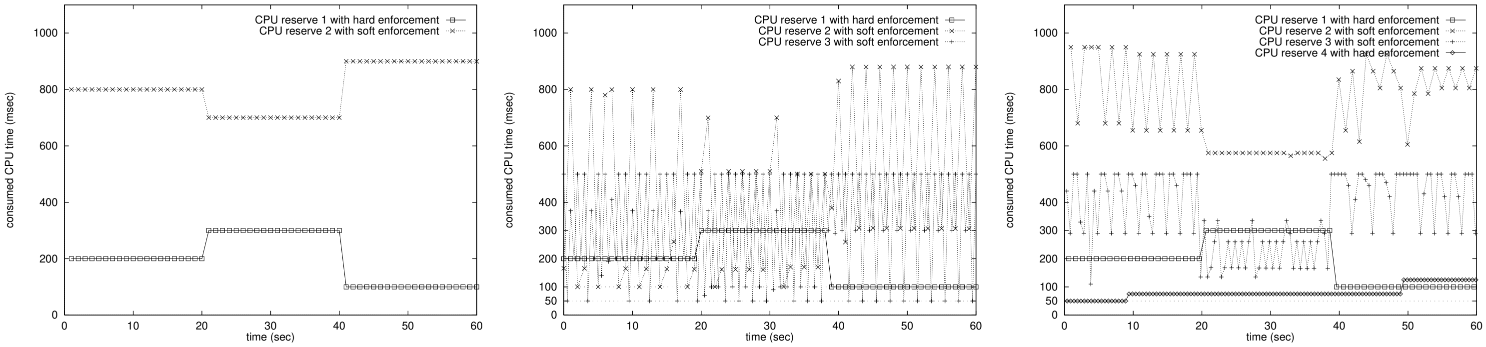

Figure 7: Enforcement of CPU Time Utilization with Reserves of Hard and Soft Enforcement

through 3 were used. The results show that processes bound to reserves with hard replenishment consume CPU times that are closest to the reserved amounts, and processes with soft replenishment consume slightly more than the specified reservation times. (Note that the time-scale on the x-axis is very fine-grained). As can be expected, the CPU times consumed by processes bound to reserves with firm replenishment are between those consumed by reserves with hard and soft replenishment. One would expect to see bigger differences between these behaviors in actual applications which make non-preemptive system calls that can take relatively long durations of time.

5 Discussions and Future Directions

In this section, we discuss the requirements for supporting Portable RK in an OS without access to the OS source code.

5.1 CPU reserves

The implementation of CPU reserves essentially requires only the following in the host operating system kernel:

- A fixed-priority scheduler,
- An exported interface to manipulate priority levels of execution objects,
- An exported interface for suspending and resuming execution objects,
- An exported interface for acquiring events within execution objects necessary for accounting reserves.

CPU Reserve	Replenishment	CPU Time (msec)	Period (msec)
1	Hard	10	500
2	Soft	10	500
3	Firm	10	500

Table 9: Task Set used for Enforcement of CPU Time Utilization with Reserves of Hard, Soft and Firm Replenishment

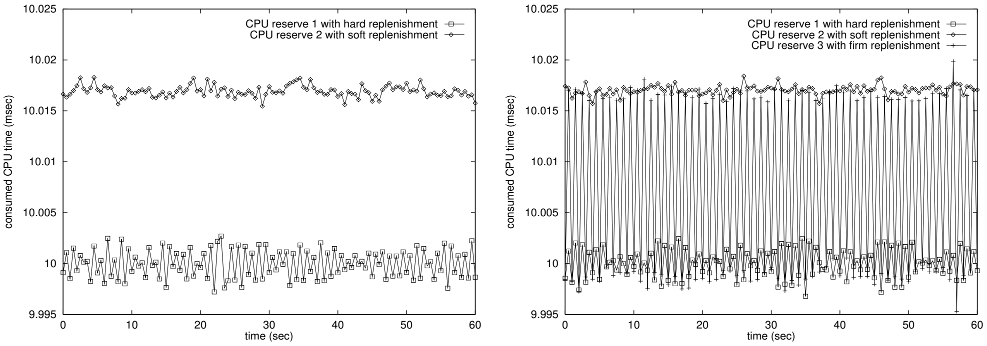

Figure 8: Enforcement of CPU Time Utilization with Reserves of Hard, Soft and Firm Replenishment

Ideally, the exported interfaces will be available from within kernel space, such that accounting and enforcement can not only be carried out efficiently but also in secure fashion. If so, Portable RK can be statically or dynamically loaded into the kernel and invoke the above interfaces to accomplish RK functionality.

Since most modern real-time and non-real-time kernels support a fixed-priority scheduler including Windows NT, Windows 95/98, AIX, Solaris, HP/UX and OS/2, it is likely that the host operating system kernel also provides the functions for manipulating priority levels of execution objects. It is also normal for these OSs to have mechanisms to suspend and resume execution objects. In particular, these interfaces are almost always available from user space. In the case of suspending and resuming execution objects, these mechanisms are necessary for device drivers to block a caller's execution until I/O data becomes available. So, mechanisms to suspend and resume threads are typically available from within kernel space.

In Linux/RK, the introduction of hooks enables acquiring events necessary for accounting reserves. This was possible because the source kernel of the Linux kernel is available. A similar feature that allows Portable RK to gain control on context-swaps between threads is available on Windows CE and the multiprocessor-versions of Windows NT [8]. This is, unfortunately, not always the case for other commodity operating system kernels. We are currently investigating the debugging features supported by the native CPU as a possible hook. Portable RK can possibly obtain the context switching information by setting a break point at the entry point to a context switching function and examine its arguments in that case.

5.2 Network Reserves

The implementation of network bandwidth reserves requires an OS interface to introduce a new queueing discipline to a subsystem that processes outgoing packets. Portable RK provides its own queueing discipline and packet scheduler to control the enqueueing and dequeueing of packets, so that enforcing the number of outgoing packets is enabled.

In Linux/RK, the Linux kernel has already implemented a packet scheduler framework and Portable RK could use it. Most UNIX variants provide STREAMS, which can intercept a packet and enables the implementation of a new packet scheduler as CBQ was implemented [2]. Similar schemes are also possible in Windows NT and Windows CE.

6 Concluding Remarks

Portable RK (Resource Kernel) is an portable implementation of a *resource kernel*, which can reside within commodity operating system kernels. Such an architecture makes it possible to experiment by using a practical software environment and helps the faster deployment of research results. As a resource kernel, Portable RK provides timely, guaranteed and enforced access to physical resources for applications that run on commodity operating system kernels.

In this paper, we described Linux/RK, which is the Linux kernel that includes Portable RK as its subsystem. The performance evaluation results showed that Portable RK brought more control over resource utilization to processes and its added overhead costs were small enough to be negligible.

We are actively working to extend Portable RK along two complementary directions. One, we are adding support for new reserve types such as network bandwidth reservation. Secondly, we are attempting to integrate Portable RK in Windows NT and Windows CE environments.

References

[1] M. Barabanov and V. Yodaiken. Introducing Real-Time Linux. *Linux Journal*, Issue 34, Feburary 1997.

[2] S. Floyd and V. Jacobson. Link-Sharing and Resource Management Models for Packet Networks. *IEEE/ACM Transaction on Networking*, Vol. 3, No. 4, 1995.

[3] C. Lee, K. Yoshida, and R. Rajkumar. Predictable Communication Protocol Processing in Real-Time Mach. In *Proceedings of IEEE Real-Time Technology and Applications Symposium*, June 1996.

[4] *http://www.linux.org/*

[5] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. *JACM*, Vol. 20. No. 1, pp.46–61, 1973.

[6] L. McVoy and C. Staelin. Lmbench: Portable Tools For Performance Analysis. In *Proceedings of Winter 1996 USENIX*, San Diego, January 1996.

[7] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. Technical Report CMU-CS-93-157, School of Computer Science, Carnegie Mellon University, May 1993.

[8] *http://www.microsoft.com/*

[9] A. Molano, K. Juvva, and R. Rajkumar. Real-Time Filesystems: Guaranteeing Timing Constraints for Disk Accesses in RT-Mach. In *Proceedings of Real-Time Systems Symposium*, December 1997.

[10] S. Oikawa and R. Rajkumar. A Resource Centric Approach To Multimedia Operating Systems. IEEE Workshop on Resource Allocation Problems in Multimedia Systems, Washington D.C., December 1996.

[11] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. A QoS-based Resource Allocation Model In *Proceedings of the IEEE Real-Time Systems Symposium*. December, 1997.

[12] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time Systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[13] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. Practical Solutions for Some QoS-based Resource Allocation Problems In *Proceedings of the IEEE Real-Time Systems Symposium*. December, 1998.

[14] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, Vol. 39, No. 9, pp. 1175–1185, 1990.

[15] S. Sommer and J. Potter. Operating System Extensions for Dynamic Real-Time Applications. In *Proceedings of Real-Time Systems Symposium*, December 1996.

[16] 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 *Proceedings of IEEE Real-Time Technology and Applications Symposium*, June 1998.

[17] P. Steenkiste, A. Fisher, and H. Zhang. Darwin: Resource Management for Application-Aware Networks. Technical Report CMU-CS-97-195, School of Computer Science, Carnegie Mellon University, December 1997.

[18] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, October 1990.

[19] I. Wakeman, A. Ghosh, J. Crowcroft, V. jacobson, and S. Floyd. Implementing Real-Time Packet Forwarding Policies using Streams. In *Proceedings of USENIX Technical Conference*, January 1995.