# Open Real-Time Linux

**Chi-sheng Shih, Jiang Qian, Mangesh Jonnalagadda**
*Department of Computer Science, University of Illinois, Urbana, IL 61801 USA*
*{cshih, j-qian, jonnalag}@cs.uiuc.edu*
**Jane Liu**
*Microsoft Corporation, Redmond WA 98052, USA*
*janeliu@microsoft.com*
**Jia-ru Li**
*Cisco Systems, Inc., San Jose, CA 95134, USA*
*juru@cisco.com*

**Abstract**

An open system allows independently developed hard real-time applications to run together with non-real-time applications and supports their reconfiguration at run-time. The open system always accepts non-real-time applications, but it never accepts a real-time application that may not be schedulable in the system. Once a real-time application is accepted, its schedulability is guaranteed regardless of the behaviors of other applications that execute concurrently in the system.

The paper describes the design and implementation of an open system in Linux and evaluates its performance. The implementation consists of three key components: a two-level kernel scheduler, a common system service provider, and real-time application programming interface (RTAPI). In Open real-time Linux, real-time applications can use existing Linux system resources and services, and there is no need to suspend the kernel in order to switch between real-time and non-real-time mode. The performance evaluation of Open Real-time Linux shows that its overhead of context switching outperforms any other available real-time linux system.

## 1 Introduction

The tremendous recent advances in hardware technologies have made it possible to run real-time applications with critical and stringent timing requirements on general purpose workstations and personal computers concurrently with non-real-time applications. This paper describes the implementation and the performance of a uniprocessor operating system that provides an open system environment to multi-threaded real-time applications. The term *open system* has been used to mean many different things. Here, we focus on the timing aspect. In effect, an open environment provides each real-time application with a slower virtual processor, isolating it from resource contention by other applications in the system. Consequently, the developer of each real-time application that is to run in an open system can choose to schedule the application according to an algorithm best suited to the application. The fact that the application can meet its real-time requirements can be validated independently of other applications.

Upon receiving a request to start a new real-time application, the open system subjects the application to a simple but accurate test that uses only a few black box parameters (e.g., required processor bandwidth and shortest relative deadline) of the application. The system accepts the application only if the application passes the test. Once the system accepts a real-time application, it schedules the application according to the algorithm chosen by application developer and guarantees the schedulability of the application (i.e., the application meets all its real-time requirements) regardless of the behaviors of other applications. In contrast, the schedulability of an application running on an existing operating system can be determined only by examining the timing attributes of every task in every application in the system. As a result, the system is closed to independently developed applications whose detailed timing attributes and resource usages are unknown.

The architecture of the open system described here

was developed by Deng, *et al.* [1, 2]. The system is implemented by extending Linux. Like the Windows NT version implemented by Deng *et al.*, the open real-time Linux extension also consists of three key components:

1. a two-level kernel scheduler and an admission mechanism, which provide timing isolation and real-time performance guarantee;
2. a prototype of service providers that deliver common system services (e.g., file server, network protocol stack handler); and
3. a set of real-time application programming interface(RTAPI) functions with which real-time applications specify their real-time attributes and communicate with service providers.

This paper describes the Linux extension that makes the operating system open. A performance evaluation study was done to determine the increase in scheduling overhead introduced by the extension and the performance of a communication server when used to support a real-time CORBA application. We present the performance data here.
Following this introduction, Section 2 discusses related works on real-time enhancements of Linux. Section 3 presents an overview of the open real-time system architecture, including the hierarchical scheduling scheme and system service providers. Section 4 describes the implementation of the open real-time Linux extension that we will call ORiS Linux[3] for short hereafter. Section 5 gives RTAPI functions and a real-time application example. Section 6 presents the results of our performance evaluation. Section 7 concludes the paper.

## 2 Related Works

There have been several projects that aim at enhancing the real-time capability of Linux. Examples are Real-Time and Embedded (RED) Linux, Real-Time Linux (RT-Linux) and Kansas University Real-Time (KURT) Linux. RED [4] provides a framework to integrate three scheduling paradigms (i.e., time driven, priority driven, and share driven) within a system. Like ORiS Linux, RED also uses a two-level scheduler. Although RED's two-level scheduler provides scheduling flexibility to fulfill different real-time requirements of different applications within a system, it does not provide timing isolation among applications. By our definition RED Linux is closed.
Real-Time Linux(RT-Linux)[5] is the first effort to support hard real-time in Linux. RT-Linux implements its own real-time kernel underneath the original Linux kernel. Non-real-time tasks are scheduled by the original Linux kernel, and real-time tasks are scheduled by the real-time kernel. RT-Linux works as the original Linux system when there is no real-time task. Once there is any real-time task in the system, the real-time kernel suspends the original Linux kernel, and hence all non-real-time tasks, to execute the real-time task. RT-Linux can isolate real-time tasks from non-real-time tasks, but not real-time tasks in different real-time applications. The only communication method between real-time and non-real-time tasks is pipe, which is inefficient. The most serious limitation is that real-time tasks cannot use any Linux kernel service.
KURT [6] provides not only a finer time resolution for Linux kernel but also different kernel modes to control the scheduling of system resources for real-time processes. Depending on the requirements of applications, the kernel can switch its scheduling mode among normal mode, mix real-time mode, and focused real-time mode at run time. When in the focused real-time mode, the system disables software interrupts and thus isolates the interferences from non-real-time applications. However, KURT provides limited scheduling support and works best when real-time applications are scheduled in a clock driven, cyclic manner.
Our work differs from these existing real-time extensions. ORiS Linux focuses on providing timing guarantees and isolation to complex, multi-threaded real-time applications, while other real-time extensions deal with the schedulability of each individual real-time task (or process). The open system allows different applications to be scheduled according to different scheduling algorithms and uses a simple but accurate acceptance test for admission control. Other real-time extensions use one scheduling algorithm to schedule all real-time tasks (or processes) and either use global schedulability analysis for admission control or provide no admission control.

## 3 Open Real-time System

Again, the open (real-time) system described here intends to provide each real-time application executing in the system with a virtual slower processor and thus isolates it from other applications in the system. For the sake of discussion here, we normalize the speeds of all virtual processors with respect to the speed of the physical processor; the speed of the physical processor is 1. To develop a real-time application, $A_k$, its developer first chooses a scheduling algorithm $\Sigma_k$ to schedule jobs (i.e., threads) in $A_k$. The schedulability of the application is then analyzed based on the assumption that the application executes alone on a slower processor of speed $\sigma_k$, which

is less than 1. The minimum speed $\sigma_k$ at which $A_k$ is schedulable is called its *required capacity* (In other words, if the execution time of a real-time job on the physical processor is $e$, the execution time used for the purpose of schedulability analysis is $\frac{e}{\sigma_k}$. ) The required capacity of every real-time application is determined by its developer. We say that the real-time application $A_k$ is schedulable in the open system if all jobs in $A_k$ meet their deadlines when $A_k$ runs together with other applications in the open system and the order in which jobs in $A_k$ are executed is determined by algorithm $\Sigma_k$.

Within the open system, the workload consists of real-time applications, called $A_1$, $A_2$, ..., $A_N$ in Figure 1, and non-real-time applications. The system uses a two-level hierarchical scheme to schedule them. All non-real-time applications are executed by the server $S_0$, while jobs of each real-time application $A_k$ are executed by server $S_k$ for $k \geq 1$. The lower level scheduler, called the *OS scheduler*, maintains and schedules all the servers in the system.

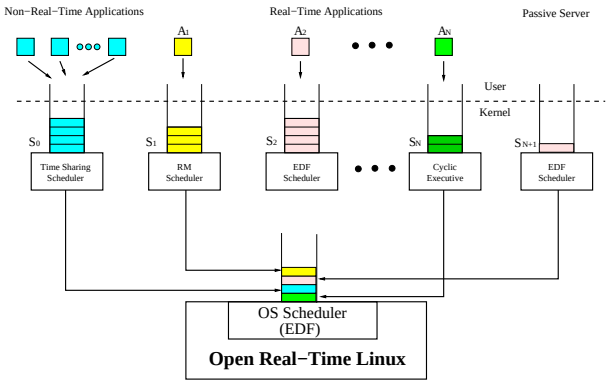



**FIGURE 1:** *Open Real-Time System*

Each server has a scheduler. The server scheduler maintains a ready queue, which contains the ready jobs of the application(s) executed by the server. When a server is scheduled by the OS scheduler, it executes the job at the head of its ready queue. Specifically, the server scheduler of $S_k$ schedules all the jobs in $A_k$ based on the scheduling algorithm $\Sigma_k$ of the application. The server scheduler of $S_0$ schedules jobs in all non-real-time applications according to the time-sharing algorithm. (In ORiS Linux, the original Linux scheduler is the server scheduler of $S_0$.) As Figure 1 shows, all the servers, server schedulers and the OS scheduler are in the kernel. Details on the principles of the two-level scheduler, the interactions between the OS scheduler and server schedulers, and the criterion used for admission test can be found in [2].

System services for network and file access are provided by system service providers, or simply service providers. Each service provider is implemented as a user level application and executes on a passive server. Unlike a server that executes an application whose size is at least equal to the required capacity of the application, passive server has a very small size, and its budget is replenished periodically. The server uses its budget only for administrative purposes. The processor time required by the service provider to perform any service is charged to the application requesting the service. This idea is borrowed from processor reserve concept by Mercer, *et al.* [7].

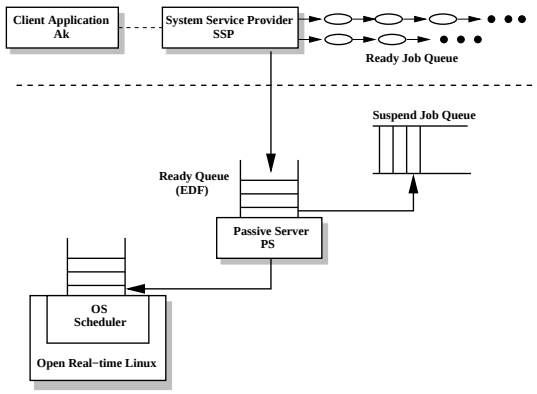



**FIGURE 2:** *System Service Provider*

Figure 2 gives an overview of the interaction among a system service provider ($SSP$), its passive server $PS$ and a real-time client application $A_k$. The application requests for a service by sending the service provider a request that is accompanied by a budget and a deadline for consuming the budget. In response to the request, the service provider creates a job $J_k$, called *work job*, to handle the request. Among the parameters of a work job are its budget and deadline, which are equal to the respective values given by the request. The job is inserted in the ready queue of the passive server $PS$ on the EDF basis according to its deadline. The budget and deadline of the passive server are equal to these parameters of the work job at the head of the queue.

# 4 Implementation

We implemented the ORiS Linux by extending the Linux 2.0.36 kernel. In this implementation, the attributes of each real-time application are stored in a KSERVER data structure and a RTtask data structure. The KSERVER declares the scheduling algorithm, required capacity, and other attributes of the real-time application. Each real-time application consists of one or more (periodic) real-time *task*s. The attributes of each real-time task are specified by its RTtask data structure. (When there is no source

of confusion, we call a real-time application a RTtask hereafter.)
A RTtask in turn consists of one or more periodic real-time *job*s. A periodic job is a task in the Linux kernel; the task is released to execute periodically. The run-time state of a periodic job is maintained by _TASK_RTEXT data structure.
There is no need to keep track of the individual tasks in non-real-time applications since the open system does not provide time isolation to non-real-time applications. These tasks are normal Linux tasks, and they are referred to as non-real-time tasks below. In the remainder of this section, we first describe new and modified data structures in the extension that implement the ORiS Linux. We then describe the modification of the kernel scheduler to support the hierarchical scheduling scheme mentioned earlier and the implementation of real-time task scheduling, task release, and system service providers.

## 4.1 Data Structures

Data structures used in ORiS Linux can be classified into three categories: data structures for OS scheduler, application server, and real-time tasks.

**OS Scheduler.** The OS scheduler is implemented as a server. The ready queue of the server maintains the run-time state of OS scheduler. The queue is defined by the data structure ReadyServerQueue. ReadyServerQueue contains a pointer to KSERVER of each eligible server (i.e., a server whose budget is not zero and its ready queue is not empty). Servers in the OS ready queue are prioritized in the order of their deadlines.
ReadyServerQueue is initialized with the non-real-time server, S0 (called $S_0$ earlier), when the system boots. S0 is a total bandwidth server [8]. The server always stays in ReadyServerQueue since we view idle task, whose task id is 0, as a ready task and therefore the ready queue of S0 is never empty and the server budget is replenished as soon as the budget becomes zero. In this way, we eliminate the cost of inserting and removing S0 from ReadyServerQueue repeatedly.

**Application Server.** KSERVER for each application server contains server attributes, run-time states, and several task queues. Server attributes includes server type, server size, required capacity, and scheduling algorithm.
In the current version of ORiS Linux, the application specifies size and type of the server when it requests a server to be created on its behalf. (In contrast, the system computes server's size and determines server's type in the NT version. We will return shortly to discuss this difference.) The OS scheduler subjects the request to an acceptance test and creates a server of the requested size if the acceptance test passes. The run-time state of a server includes the server ID, next event time, current scheduled job, budget, and deadline. Like other schedulers, a server scheduler for a real-time application maintains several queues for tasks executing on a server. These queues are task list, ready queue, and suspend queue. A server is destroyed when the last task in its task list terminates.

**Real-Time Extension.** To support real-time applications in ORiS Linux, we modified the run-time task state data structure, task_struct, and added two data structures. The new data structures are used to maintain the attributes of real-time tasks and jobs; they are called RTtask and _TASK_RTEXT.
In Linux kernel, data structure task_struct declared in include/linux/sched.h keeps the task state information, including task priority, user id, and timer list. Three variables are added to augment this structure; they maintain the task state information for real-time jobs.

```
struct task_struct{
        /*Original task structure definition*/
        ...
        /*Real-time task structure definition starts*/
        struct _TASK_RTEXT *TaskRTExt;
        struct task_struct *RTnext_task, *RTprev_task;
}
```

Variable TaskRTExt is a pointer to the real-time extension data structure of a real-time job. RTnext_task and RTprev_task are used to maintain a list of real-time jobs executed by a server. These three variables are NULL for non-real-time tasks.
As discussed earlier, RTtask is a data structure used to keep the attributes of a real-time task. Within RTtask, we declare variables for task's id, period (i.e., the length of the time interval between the releases of consecutive jobs of the task), relative deadline (i.e., the maximum allowed response time of jobs in the task), release phase (i.e., the release time of the first job in the task), etc.

```
typedef struct _KRTTASK{
        LARGE_INTEGER       id;
        /* LARGE_INTEGER is defined as an
                            unsigned long integer.*/
        LARGE_INTEGER       period;
        LARGE_INTEGER       relativeDeadline;
        LARGE_INTEGER       executionTime;
        ...
}RTtask;
```

Similar to real time task extension, real-time job extension TASK_RTEXT keeps track of the state of each real-time job. The variables in this data structure provide values of the job's remaining execution time, release time, deadline, pointer to its server, and so on. Part of the data structure declaration is shown below.

```
typedef struct _TASK_RTEXT{
        LARGE_INTEGER       priority;
        LARGE_INTEGER       releaseTime;
        LARGE_INTEGER       remainingExecTime;
        LARGE_INTEGER       deadline;
        ...
}TASK_RTEXT;
```

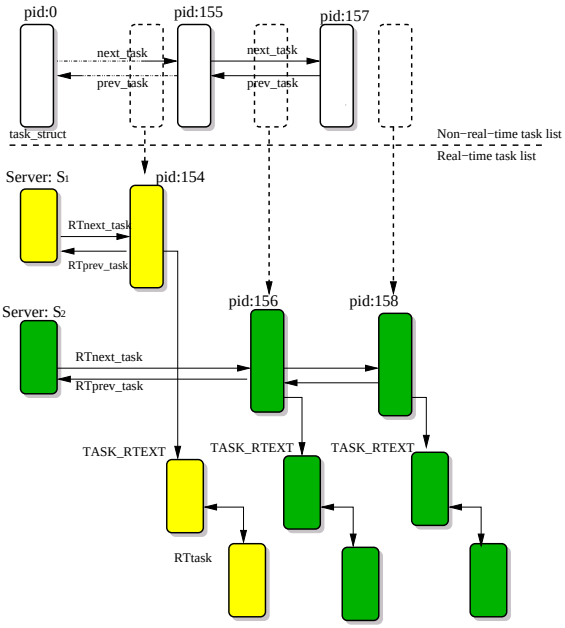


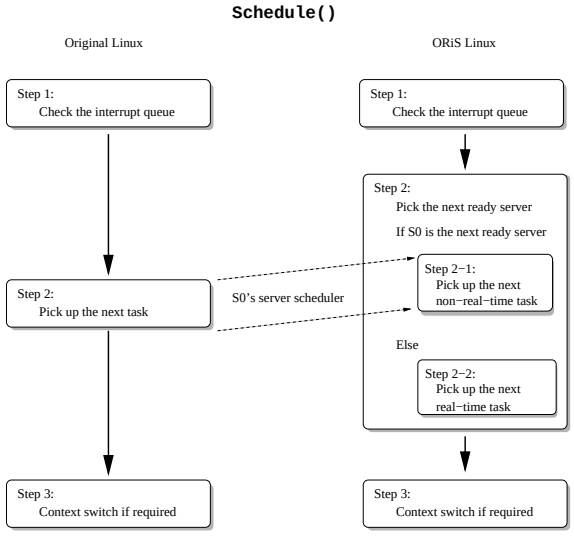

**FIGURE 3:** *Task list for non-real-time and real-time tasks*

Figure 3 gives an example to illustrate the task lists maintained by the system. In the upper portion, we show the task list of non-real-time tasks. This double-linked list starts with idle task with pid 0 and contains all non-real-time tasks in the system. This is the task list maintained by the original Linux kernel. We therefore call it Linux task list. In this example, there are two real-time servers, $S_1$ and $S_2$ (i.e., two real-time applications.) There is one RTtask within server $S_1$ and two RTtasks within server $S_2$. Each RTtask has only one real-time job. Dashed rectangles in this figure represent the task_structs of real-time jobs. When a real-time application creates a real-time job by calling system call RT_add_job(), a RTAPI function which we will describe later in this section, its task_struct is removed from the Linux task list and appended to the task list of the server of the calling real-time application. As a consequence, real-time jobs will not be scheduled by the Linux scheduler, $S_0$. Task list of each server is linked by RTnext_task and RTprev_task pointers instead. Although task_struct of real-time jobs have been removed from the task list of $S_0$, array task[] keeping the pointers to all tasks in the system still keeps the pointer to real-time jobs. Therefore, the real-time jobs are visible via ps command.

We use two data structures,TASK_RTEXT and RTtask, to keep track of real-time attributes of each task [1]. The reason is the relative deadline of the job may be greater than task's period. If this is the case, we can have more than one real-time job for the real-time task at the run time.

## 4.2 Two-Level Scheduler

Like other desktop operating systems, Linux uses a time sharing scheduler to schedule tasks. This scheduler is implemented by function scheduler() in linux/kernel/sched.c. Function scheduler() takes three steps to schedule a task in the way shown in Figure 4. Linux kernel calls function scheduler() to find the next task when the current executing task completes or is blocked or a clock interrupt occurs. When function scheduler() starts, it first checks the interrupt queue to see if there is any queued interrupt and executes the interrupt service routine if there is any. The second step finds the next ready task to execute by comparing the remaining quota[2] of each task on the task list. The third step does the context switch if the next ready task is not the current task.




[1] The current implementation assumes that the relative deadline is no greater than the task period.

[2] Linux kernel gives each task 1000 unit of clock tick when a task starts. The quota decreases by 1 for each clock tick when the task executes.

**FIGURE 4:** Schedule() *function in original Linux and ORiS Linux*

In ORiS Linux, the first and third step are unchanged. Within the second step of OS scheduler, the scheduler finds the next ready server from the ready server queue first. The server scheduler of the next ready server is then called to find the next ready task. If the next ready server is $S_0$, the second step of the original Linux is used to find the next ready non-real-time task. Otherwise, if the server is a real-time server, the scheduler of the server is called.

Two level scheduler

```
asmlinkage void schedule(void) {
  /* Step 1: Check interrupt queue */
   ...

  /* Step 2: Find the next ready server */
  nextSer = RTGetHighPriorityServer();
                                              /* Pick the next server*/

  if (nextSer == S0)
    then                                      /* Schedule non-real-time tasks*/
        /* Step 2 of original schedule(): find the */
        /* next task to run */
         ...
    else                                      /* Schedule real-time tasks*/
        next = nextSer->getNextJob(nextSer);

  /* Step 3: Context Switch */
   ...
}
```

**FIGURE 5:** *Two Level Scheduler*

The modification of function schedule() is shown in Figure 5. After the first step, the OS scheduler finds the next ready server by calling function RTGetHighPriorityServer(). RTGetHighPriorityServer() returns the highest priority server in the ready server queue, ReadyServerQueue. Since EDF policy is used, the highest priority server is the server with earliest deadline. If the next server is the non-real-time server, S0, the second step of original Linux scheduler is executed. Otherwise, real-time server scheduler, getNextJob(), is called to find the next ready real-time job from its ready queue.

Within real-time server, KSERVER, function pointer variables getNextJob and extractNextJob point to the functions to find and remove the next ready real-time job from server's ready queue. When the system creates the server, it assigns these two functions according to server's scheduling algorithm. For instance, the function to find the next ready task for Rate Monotonic algorithm[9] is implemented as function getNextRMJob() in linux/kernel/rm.c. Suppose that jobs executed by a real-time server S1 are scheduled according to the Rate Monotonic algorithm. The function pointer variable is set by using

S1->getNextJob = getNextRMJob.

Whenever function S1->getNextJob() is called, function getNextRMJob executes instead. Thus we eliminate the time spent to check to the type of scheduling algorithm of each server to call different scheduling function.

## 4.3 Real-time task Scheduling

Specifically, the scheduler for each server is implemented by two functions, getNextJob and extractNextJob. getNextJob finds the next ready job on server's ready queue, returns the pointer of the job's task structure, and leaves the job on the ready queue. Function extractNextJob is only called when the job completes. The function not only returns the pointer but also removes the job from the ready queue. The extractNextJob function then appends the newly removed job to the suspend queue of the server if the job is to be released later to run again (i.e., there are more instances to run.) Otherwise, it removes the job from the system.

For a rate monotonic server (i.e., a server whose scheduler uses the rate-monotonic algorithm) and other fixed priority servers, 256 FIFO queues with distinct priority are used to store the ready tasks. Priority of each real-time job is mapped to one of 256 priorities by constant ratio mapping[10]. The minimum and maximum rate for rate monotonic scheduling is given in linux/include/linux/rm.h by MinRate and MaxRate respectively. Function getNextRMJob, which is the getNextJob function for rate monotonic server, uses a bit string to keep track of the backlogged queues. The job at the head of the highest priority queue is then chosen as the next ready job.

## 4.4 Task Release

Real-time jobs are released by timers. After a real-time task is moved to the task list of the server that will execute it, the function RT_Start_job() sets the timer to release the job of the task according to the phase of the task. Subsequently, the timer callback function RT_job_timeout(), which is implemented in linux/arch/i386/process.c, sets the timer to release latter jobs in the task periodically.

Since Linux kernel only checks its timer list at clock ticks which are 10 milliseconds apart on x86 platform[3], the timer callback function may not be invoked precisely at its expiration time unless the release time coincides with a clock tick. The inaccuracy of the timing of timer callback invocation affects both the acceptance test of real-time applications and the implementation of the two-level kernel scheduler. As for the acceptance test, tick size (i.e., the period of clock ticks) is treated as an extra blocking time. Similarly, the two-level scheduler is implemented as a

[3]Tick size is 1 millisecond on Alpha platform

tick scheduler: new threads are added to the scheduler's ready queue and preemption can occur only at each tick time. As a result, when a job is released, it can be blocked for as much as the duration of the tick size. The OS scheduler must give the real-time application a larger server size than its required capacity in order to mask the effect of this blocking[2]. This results in a lower processor utilization.
Also, we had to take special care in implementing the two-level scheduler to correctly handle the job releasing and server deadline expiration. If a job is released in the same tick interval as the the deadline of the server executing the job, we must make sure that the job's release timer callback function is invoked before the deadline expiration timer callback function. Otherwise, if the deadline expiration callback function were executed first, the OS scheduler would replenish the server before new job is inserted in the server ready queue. As a consequence, the amount of budget replenished would likely be wrong, moreover, a wrong job might be executed by the server.
In Linux kernel, if two or more timers expire within same tick interval, the later its expiration time, the earlier its timer callback function is invoked. To ensure the correct invocation order of the timer, we always adjust the expiration time of each job release timer towards the end of its expiring tick interval, and the expiration time of each server deadline timer towards the beginning of its expiring tick interval. For the same reason, when a job release timer callback function is invoked, we need to find if there are other job release timers of the same server also expiring in that tick interval. The OS scheduler replenishes the budget for the server only in the last job release timer callback function invoked for the tick interval.

### 4.5 System Service Providers

Again, system service providers(SSP) in the open system are user-level applications executing by a passive server. A communication server is implemented in the ORiS Linux. In the current version, the architecture of SSP is as described in Section 3.2. (We will show latter in Section 6.2 that this architecture introduces overheads in creating jobs and redundant initialization for each resource request. A new architecture that keeps these overheads small is proposed in that section.)
The interaction between a client(a real-time job) and the communication server is shown in Figure 6. Specifically, two function calls, mesg_recv and mesg_send, are provided for sending and receiving data via TCP. Real-time client jobs and the administrative jobs of the server communicate via message queue and share memory. On the communication server, two administrative jobs, sendWatchingThread and recvWatchingThread, are created when the communication server starts. These two administrative jobs listen on the message queue to receive requests from clients.

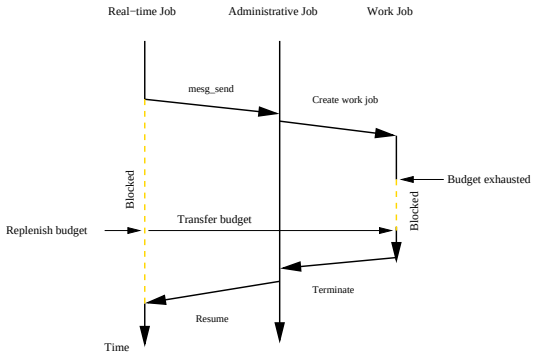



**FIGURE 6:** *Interaction between real-time job and communication server*

When the real-time job sends a request to the communication server, its remaining budget and deadline associated with the budget are sent along with the request, and the budget of client's server is set to 0. The server and hence the client job are suspended. Upon receiving the request, the administrative job of the communication server creates a work job to execute on the client's behalf. The budget and deadline of the passive server used to execute this work job are set to the remaining budget and deadline passed by the client. Work job executes till the task completes or the budget is exhausted. In the latter case, the budget of the passive server will be replenished at the budget replenishment time of the client application's server so that the execution of the work job can continue. On the other hand, if the task completes while the passive still has budget, the remaining budget is discarded in the current implementation. In other words, we assume there is no further work after the request. Often, further work needs to be done after the work job completes. In the current version, such a real-time task is split into more than one task. Further work can be completed by carefully arranging the release phases of the spilt tasks.

## 5 Real-time Application Interface and application framework

ORiS Linux provides a set of application interface for real-time applications. This section describes the application interface functions and gives an example of real-time applications.

## 5.1 Real-time Application Interface

A real-time application uses RT_init_server(KSERVER *server) to request admission, that is, to execute as a real-time application. In the current version, the argument server declares the required capacity of the application, the scheduling algorithm and server type. When called by this function, the OS scheduler to carry out an acceptance test. The current version of ORiS Linux supports only periodic tasks without release time jitter and without nonpreemptable section. Under this restriction, the server size needed to guarantee schedulability is equal to the required capacity of the applications. The acceptance test simply checks whether 1 minus the total size of all existing servers is no smaller than the required capacity. If the application passes the test, the system creates a server and returns the id of the server. Otherwise, "-1" is returned to reject the request.

As mentioned earlier, server size and server type are determined by the system in the NT version. This is necessary because that version also supports real-time applications that may content for global resources and hence may have nonpreemptable section. When some parts of real-time jobs are nonpreemptable, the server size required to ensure the schedulability of each real-time application also depends on the maximum execution time of nonpreemptable sections of other real-time applications in the system. The maximum execution time of all nonpreemptable sections (i.e., the length of time any real-time job is allowed to be nonpreemptable) is also a parameter provided by each application when it requests to execute as a real-time application.

The function RT_add_job(void (*fn)(void *), void *data , RTtask *rttask) is called to add a task to the task list of a server. Task's attribute is specified in rttask. Specifically, rttask gives the ID of the server, which is returned by the system after the application containing this ask passed its acceptance test. The period, maximum execution time, ready time, relative deadline, and number of instances of the real-time task are declared there. The function pointer *fn() points to the function executed by the real-time job.

RT_clone() is called by function RT_add_job() to clone the task that executes a real-time job. ORiS Linux also subjects each real-time task to an acceptance test, and thus, independently validates that the task is indeed schedulable on a virtual slower processor of speed equal to the server size. Acceptance test is done according server's reserved capacity and scheduling algorithm, as well as task's instantaneous utilization. When called, RT_clone first carries out an acceptance test. In order to create the task for a real-time job, RT_clone() not only calls do_fork() to create the new task but also initializes real-time extension data structures. After calling do_fork(), the parent task removes the child task from system's ready queue to prevent the child task being scheduled by non-real-time scheduler and puts the child task into server's suspend queue.

The function RT_start_job(int SerId) is called to start the real-time applications on real-time server SerId. RT_start_job() is asynchronous.

At the end of each instance of real-time job, RT_rewind() is called to determine whether the system should release another instance of the real-time job. If the current real-time job is not the last instance of the real-time task or the real-time task has infinite instances, kernel sets a timer at the next release time.

## 5.2 Real-time Application Example

Figure 7 shows an example of real-time application. The required capacity is one half. This application consists of one or more task and is scheduled by the rate monotonic algorithm.

Real-Time Application Example

```
main()
{
    int SerId;
    TASK_RTEXT rtext;
    KRTTASK tmp, tmp2;
    KSERVER *server;

     // Allocate memory for server and RTtask
                                                    /* Setup Real-time Application Server */
    server->type = ConstantUtilizationServer;
    server->algorithm = RateMonotonic;
    server->size = 5000;
                                                    /* Acceptance Test */
    SerId = RT_init_server( (KSERVER *) server );
    if (SerId == -1){
      fprintf(stderr, "RT_init_server() fails");
      return;
    }
                                                    /* Declare Real-time Application Task */
    memset(RTtask, 0, sizeof(KRTTASK));
    RTtask->id = SerId;                             /* Assign Server ID */
    RTtask->period = 4;                             /* Period length(ticks) */
    RTtask->executionTime = 2;
                                                    /* Maximum Execution Time */
    RTtask->readyTime = 10;
                                                    /* Real-time task's ready time*/
    RTtask->relativeDeadline = 4;
                                                    /* Must less or equal than period*/
    RTtask->totalInstance = 256;                    /* Number of instance.*/
                                                    /* Create a real-time job */
    ret = RT_add_job(RealTimeJobFunction_1,
                (void *) NULL, (PKRTTASK)(RTtask));

    /* Create other real-time tasks */
    ......
    /* Start Real-time application */
    ret = RT_start_job( SerId );
}

/* Declare the job content of real-time task */
void RealTimeJobFunction_1(void *data){
   int ret;
   for (;;)
     /* Do whatever you want here */
     ret= RT_rewind();                              /* Check next release */
     if (!ret)     break;
   }
}

/* Declare the job content of other real-time tasks */
......
```

**FIGURE 7:** *Real-time Application Example*

After a constant utilization server with size 5000 (i.e., $\frac{1}{2}$) and the rate monotonic algorithm are set for the server, the attributes of the first real-time task are

specified in the declaration of the RTtask structure of the task. In this example, it is a periodic task whose period is 40 milliseconds, i.e. 4 ticks, and whose maximum execution time is 20 milliseconds. Its relative deadline is 40 milliseconds. The task has 256 instances. (When TotalInstance is 0, the real-time task will execute till the task is terminated using kill command.) The function RT_Add_job is then called to specify the starting address of a real-time task and create real-time task's data structure RT-task. Each instance of the first real-time task executes the function RealTimeJobFunction_1() function. The function RT_Add_job can be called more than once to add other real-time tasks into a real-time application server. By giving the id of the real-time application server, function RT_start_job() starts the real-time application.
RealTimeJobFunction_1 defines the work of the real-time job in the first task. Real-time job is encapsulated by an infinite loop. At the end of each instance of real-time job, function RT_rewind() is called to determine if there is any more instance to run. False value of RT_rewind() indicates the end of the real-time task and terminates the loop.

# 6 Performance Evaluation

We conducted a two-part study to evaluate the performance of ORiS Linux. In the first part, we investigated the scheduling overhead of ORiS Linux and compared it with that of the original Linux. By the original Linux we mean RedHat Linux version 5.2 with 2.0.36 kernel. In the second part, we evaluated the performance of a real-time CORBA application on ORiS Linux. The performance measure used for this purpose is completion rate, which is the ratio of tasks meeting deadlines to total number of tasks.
A Pentium II 266 PC with 64 MB RAM and 8 GB disk driver was used for the first part of our evaluation. Another PC equipped with Pentium Pro 200 processor and 64 MB RAM was used as a server in the second part.

## 6.1 Scheduling Overhead

Our modification of the Linux task scheduler introduces extra (scheduling) overhead. In particular, the ORiS Linux's task scheduler checks the head of the ready server queue for the next ready server no matter whether there is any real-time application or not. If the next ready server is S0, the server for non-real-time applications, the original Linux's time sharing scheduler is called. To determine the extra overhead thus introduced, we compared the overhead of original Linux's task scheduler with that of the ORiS Linux's task scheduler. We measured the overhead by the length of time required by step 2 in Figure 4, the step in which Linux and ORiS Linux operates differently. Step 2 of function schedule() checks the remaining quota of all tasks in the task list and picks up the task with maximum remaining quota. Therefore, the length of the task list affects the overhead of step 2. In order to eliminate the impact of the length of task list, we measured the overhead when the system starts. Therefore, we can have the same workload and can keep the task list as short as possible.
The distribution of this overhead is shown in Figure 8. We can see that the peak of overhead distribution moves from 0.2 microseconds to 0.4 microseconds. Specifically, the extra overheads introduced by ORiS Linux are approximately 0.2 microseconds. The length of time the ORiS Linux's task scheduler takes to complete Step 2 is approximately time as long as the time taken by the original Linux's task scheduler. However, task scheduling only happens at every tick or when rescheduling is required. Comparing to the tick size, 10 milliseconds in Linux, 0.2 microseconds is small.

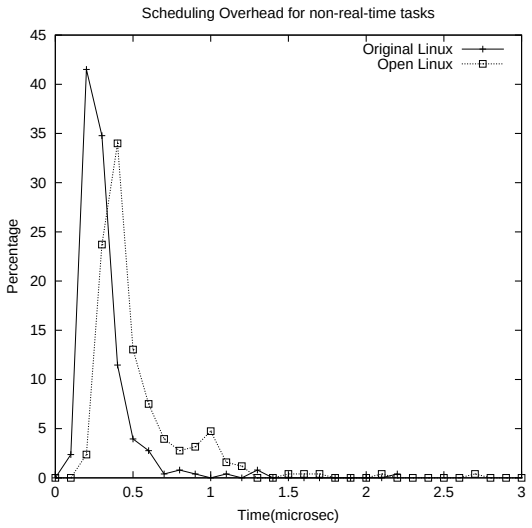



**FIGURE 8:** *Scheduling Overhead for non-real-time tasks*

Comparing to the scheduling overhead of non-real-time tasks, the overhead of scheduling real-time tasks is much larger. Original Linux's scheduler finds the next ready task and does the context switch. By scanning the task list, original Linux picks up the task with maximum remaining quota as the next task. On the other hand, the scheduler of a real-time server needs to find the next ready job from a set of queues instead of one queue or list. Once the highest priority queue is founded, the scheduler picks up the job at the head of the queue and maintains the queue in the correct order. For instance, the scheduler of a

server using the rate monotonic algorithm needs to find the highest priority queue among the 256 FIFO queues. This overhead occurs when the next ready server is a real-time server only.

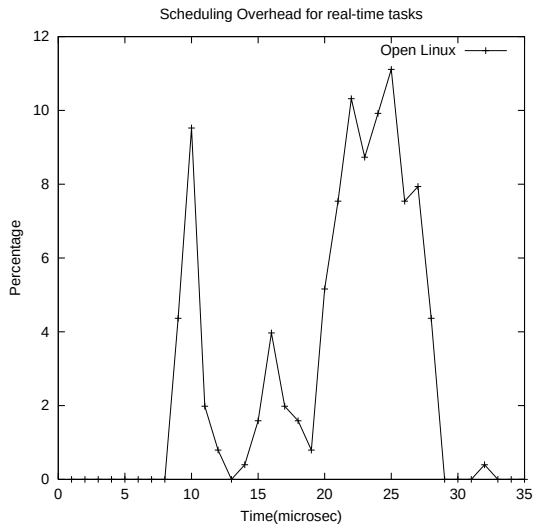



**FIGURE 9:** *Scheduling Overhead for real-time tasks*

The evaluation result is shown in Figure 9. In this evaluation, we executed a real-time task whose period is 20 milliseconds and execution time is 10 milliseconds. 3000 instances of this real-time task are executed. We can see that most of overhead falls into the range from 15 microseconds to 30 microseconds. Comparing to other Linux real-time extensions, ORiS Linux has the smallest overhead. (For example, the scheduling overhead introducing by RED is around 1,700 microseconds when the tasks execute on a Pentium II 400 machine[4].)

## 6.2 Performance of System service provider

A communication server that provides TCP connections is implemented in the current version. To investigate the performance of the communication server, a real-time CORBA application was used to see if ORiS Linux can finish jobs on time when the jobs access the network.

The application contains a real-time client task and a server task. The client task has periodic jobs that are released for every 20 milliseconds and each job sends a message to the server to request data and waits for the result. The server keeps listening to serve request messages. The client task executed on one machine with Pentium II 266 processor, and the server task executed on another machine with Pentium Pro 200 processor. The required capacity for the client task is one half, i.e., it executes 10 milliseconds for each 20 millisecond period. (Again, we assume the relative deadline is equal to its period.) ORBacus[11] and TAO[12] were used as the non-real-time and real-time ORB respectively. The client tasks on ORiS Linux were implemented with ORBacus and those on the original Linux were implemented with ORBacus and TAO.

We investigated the completion rate under various environments. The completion rate is defined as the ratio of the number of tasks finishing on time to the total number of tasks. Specifically, the scheduling framework with 100% completion rate finishes all jobs on time. To investigate the completion rate under different system workload, a periodic task enforcing workload is used to consume computation bandwidth. This periodic task provides 10%, 20%, 30%, 40%, and 45% workload. The result is shown in Figure 10.

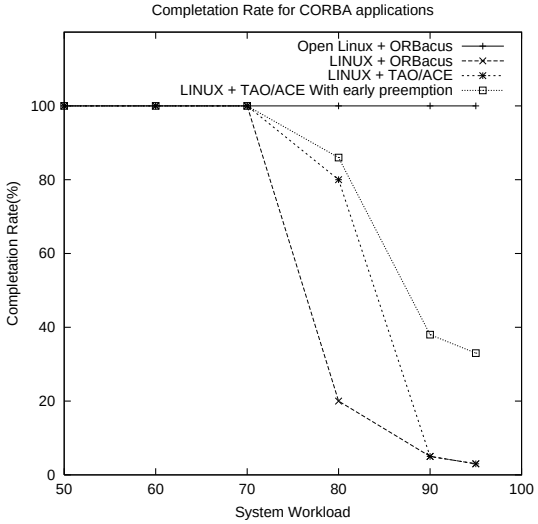



**FIGURE 10:** *Completion Rate Comparison for CORBA Applications*

The minimum system workload shown in Figure 10 is 50%. This 50% workload comes from the CORBA periodic task that executes 10 milliseconds for each 20 millisecond period. ¿From Figure 10, we can see clearly that the CORBA periodic task on ORiS Linux can always finish on time no matter how heavy the system workload is. On the other hand, once the system workload exceeded 70%, it started missing deadlines when it ran on the original Linux. This result shows that the system service provider can manage the system service well.

During the evaluation, we found that the system service provider introduces high overhead. In current design, the work job sending or receiving data on behalf of a real-time job is created by the administrative job of the passive server, not by the real-time job. Therefore, the work job does not inherit the task attributes of the real-time job including file handler

table and variables. In such a circumstance, the work job needs to make a new connection for each call. This leads to unnecessary initialization cost. The overhead to create a new task is also unnecessary.
An alternative is to execute the communication task by the real-time job itself. In this case, we move the real-time job from its application server to the passive server. When the communication task finishes, the real-time job is returned to the original server. With the reorganization, we can eliminate the cost of creating a new task and making a new connection for each transmission.

# 7 Summary

This paper gives an overview of an open system environment for real-time applications and describes its design and implementation in Linux, called ORiS Linux. To implement ORiS Linux, we replaced the existing Linux kernel scheduler by a two-level kernel scheduler, prototyped a system service provider as a user-level application, and provided a set real-time application programming interface. ORiS Linux allows independently developed real-time applications to run on Linux together with non-real-time applications. The schedulability of each real-time application can be determined without global schedulability analysis. Once ORiS Linux admits a real-time application, it provides the application with timing guarantees.

# References


[1] Z. Deng and Jane W.-S. Liu, "Scheduling real-time applications in an open environment," in *Proceedings of Real-Time Systems Symposium*, (San Francisco, California), pp. 308–319, IEEE, December 1997.

[2] Z. Deng, Jane W. S. Liu, L. Zhang, S. Mouna, and A. Frei, "An open environment for real time applications," *Real-time Systems*, vol. 16, pp. 155–185, May 1999.

[3] Real-time Systems Laboratory(RTSL), Department of Computer Science, University of Illinois at Urbana-champaign, "ORiS Linux – An Open Real-Time Scheduling Linux." http://pertsserver.cs.uiuc.edu/~cshih/ORIS.

[4] Yu-Chung Wang and Kwei-jay Lin, "Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel," in *Proceedings of IEEE Real-Time Systems Symposium*, 1999.

[5] M. Barabanov and V. Yodaiken, "Introducing Real-Time Unix," *Linux Journal*, Feb. 1997.

[6] B. Srinivasan et al, "A firm real-time system implementation using commerical off-the shelf hardware and free software," in *Proceedings of IEEE Real-Time Technology and Application Symposium*, pp. 112–119, 1998.

[7] C. W. Mercer, S Savage, and H Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, IEEE, May 1994.

[8] M. Spuri and G. Buttazzo, "Scheduling aperiodic tasks in dynamic priority systems," *Real-Time Systems Journal*, vol. 10, pp. 179–210, 1996.

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

[10] John P. Lehoczky, Lui Sha, and Ye Ding, "The rate monotonic algorithm: Exact characterization and average case behaviour," in *Proceedings of IEEE Real-Time Systems Symposium*, pp. 166–171, IEEE, 1989.

[11] Object Oriented Concepts, Inc., "ORBacus." http://www.ooc.com/ob/.

[12] Distributed Object Computing(DOC) Group, Department of Computer Science, Washington University, "TAO." http://www.cs.wustl.edu/ schmidt/TAO.html.