JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 22, 31-47 (2006)

Real-Time Linux with Budget-Based Resource Reservation*

NEI-CHIUNG PERNG, CHIN-SHUANG LIU AND TEI-WEI KUO
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, 106 Taiwan
E-mail: {d90011; p89010; ktw}@csie.ntu.edu.tw

The purpose of this paper is to propose a budget-based RTAI (Real-Time Application Interface) implementation for real-time tasks over Linux on x86 architectures, where RTAI provides a light-weight, high-performance interface for hard and soft real-time tasks over Linux. Our revised RTAI API's are extended to enable programmers to specify a computation budget for each task, and backward compatibility is maintained with the original RTAI design. Different from the past work, we focus on the implementation of budget-based resource reservation for real-time tasks, which is made complicated by the relationship between RTAI and Linux. Modifications of RTAI are limited to a few procedures without any change made to the Linux source code, such as the timer interrupt handler, the RTAI scheduler, or rt_task_wait_period(). The feasibility of the proposed implementation is demonstrated by a system operating under Linux 2.4.0-test10 and RTAI 24.1.2 on PII and PIII platforms.

Keywords: RTAI, Linux, QoS, budget reservation, real-time Linux

1. INTRODUCTION

Various levels of real-time support are now provided in most commercial operating systems, such as Windows XP, Windows CE .NET, and Solaris. However, most of them only focus on non-aging real-time priority levels, interrupt latency, and priority inversion mechanisms (merely at very preliminary stages). Although real-time priority scheduling is powerful, it is a pretty low-level mechanism. Application engineers might have to embed mechanisms at different levels inside their codes, such as those for frequent and intelligent adjustment of priority levels, or provide additional (indirect management) utilities to fit the quality-of-services (QoS) requirements of each individual task. In the past decade, researchers have started exploring scheduling mechanisms that are more intuitive and better applicable to applications, such as budget-based reservation [1-6] and rate-based scheduling [7-9].

Most commercial operating systems now claim the support for real-time applications. VxWorks [10] provides efficient preemptive scheduling, deterministic context switching time, and swift interrupt handling, and is used in many mission-critical applications ranged from anti-lock braking to inter-planetary exploration. pSOSystem 3 [11] is designed for quick development of embedded devices. Its pSOS+ 3 multitasking kernel

Received July 4, 2005; revised October 3, 2005; accepted October 10, 2005.
Communicated by Kwei-Jay Lin.
* This research was supported in part by the National Science Council under grants NSC93-2752-E-002-008-PAE and NSC93-2218-E-002-140.

is small, fast, reliable, and deterministic. RTAI (Real-Time Application Interface) proposed by Mantegazza *et al*. [12] at DIAPM shares a very similar system architecture with RTLinux proposed by Yodaiken *et al*. [13]. All interrupts are intercepted by a tiny kernel under Linux such that hard real-time tasks are supported by the tiny kernel, where Linux is considered as a "process" for the tiny kernel. BlueCat RT from LynuxWorks adopts RTLinux inside its RT kernel. Embedix Realtime, i.e., Lineo's real-time Linux distribution, is derived based on RTAI. MontaVista provides a fully preemptible Linux kernel, where kernel-mode tasks could be interrupted, and rescheduling is possible. REDICE-Linux [6, 14] delivered by RedSonic represents a hybrid real-time Linux solution, which revises Linux and integrates RTAI inside its kernel. One integrated kernel is used to schedule all real-time and non-real-time tasks. Solaris 8 supports high resolution timers and user-level priority inheritance, where priorities from 100 to 159 are belonging to real-time tasks, priorities from 160 to 169 are for interrupt servicing threads, and the rest are for ordinary user tasks. Windows CE .NET provides 256 non-aging real-time priority levels and bounds the interrupt service routine (ISR) latency within 1 *ms*. A very preliminary one-level priority inheritance is provided to allow a task to inherit the priority of a higher-priority task blocked by the former task. RTX from VenturCom [15] is widely adopted by many Windows-family operating systems and vendors to provide high-performance real-time support. A preemptive scheduling policy with priority promotion to prevent priority inversion is provided.

The concept of budget-based reservation, that is considered as an important approach for applications' QoS support, was first proposed by Mercer *et al.* [3, 4, 16]. A microkernel-based mechanism was implemented to let users reserve CPU cycles for tasks/threads. Windows NT middlewares [1, 2] were proposed to provide budget reservations and soft QoS guarantees for applications over Windows NT. REDICE-Linux implemented the idea of hierarchical budget groups to allow tasks in a group to share a specified amount of budget [5]. There were also many other researches and implementation results on the QoS support for real-time applications, e.g., [17-21]. In particular, Adelberg *et al.* [22] presented a real-time emulation program to build soft real-time scheduling on the top of UNIX. Childs and Ingram [23] chose to modify the Linux source code by adding a new scheduling class called SCHED_QOS which let applications specify the amount of CPU time per period[1]. Abeni *et al*. presented an experimental study of the latency behavior of Linux [24]. Several sources of latency was quantified with a series of micro-benchmarks. It was shown that latency was mainly resulted from timers and non-preemptable sections.

The purpose of this paper is to explore budget-based resource reservation for real-time LXRT tasks which run over Linux and propose its implementation. We first extend RTAI API's to provide hard budget guarantees to hard real-time tasks under RTAI. We then build up an implementation for soft budget guarantees for LXRT tasks over that for hard real-time tasks under RTAI. Backward compatibility is maintained for the original RTAI (and LXRT) design. We present our software framework and patch for the proposed implementation. We try to minimize the modifications on RTAI without any change to the Linux source code. Modifications on RTAI are limited to few procedures, such as the timer interrupt handler, the RTAI scheduler, and rt_task_wait_period().

[1] This approach is very different from that reported in this paper. We propose an RTAI-based implementation that can provide budget-based reservation for hard and soft real-time applications.

The rest of the paper is organized as follows: Section 2 summarizes the layered architecture and the functionalities of RTAI. Section 3 presents our motivation for this implementation design. We extend RTAI API's to provide hard budget guarantees to hard real-time tasks under RTAI and then propose our implementation for soft budget guarantees for LXRT tasks over that for hard real-time tasks under RTAI. Imprecision issues of budget-based resource reservation for LXRT tasks are also addressed. Section 4 provides the overhead evaluation of the proposed implementations, and we conclude in section 5.

2. REAL-TIME LINUX APPLICATION INTERFACE

RTAI enables Linux to achieve deterministic and preemptive performance for hard real-time tasks. The layered architecture of RTAI [25] is shown in Fig. 1 (a): RTAI basic modules related to hardware control, RTAI_SCHED (the real-time scheduler), RTAI_ FIFO (FIFO's and semaphores), RTAI_SHM (shared memory support), LXRT (the advanced module for user-mode Linux tasks that invokes RTAI API's), RTAI_ PTHREAD.O (POSIX 1003.1c support), RTAI_PQUEUES.O (POSIX 1003.1b message queues support), and RT_MEM_MGR (dynamic memory management).

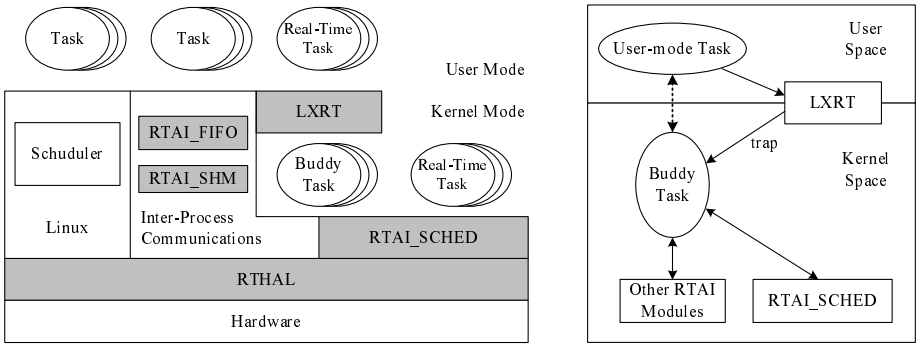


Fig. 1. (a) The RTAI architecture. (b) The relationship between a real-time LXRT task and its buddy task under RTAI.

Real-Time Hardware Abstraction Layer (RTHAL), which is an interface between the hardware and Linux kernel, consists of a couple of data structures. With RTHAL, all hardware interrupts are intercepted and routed to either RTAI handlers or standard Linux handlers. This architecture enables RTAI to have good interrupt latency and short context switching time for hard real-time tasks[2]. The RTAI_SCHED module is the layer used for real-time scheduling. Scheduling can be driven by the timer (rt_timer_handler()) or requested voluntarily by tasks through task suspending (rt_task_wait_period()). Real-time tasks scheduled by RTAI always have priorities higher than others of the Linux

[2] RTAI offers a context switch time of 7 μs, 15 μs interrupt latency, and 30 KHz one-shot task rates on PII platforms.

kernel. In other words, the Linux kernel runs just like an idle task, executing while the system is idle, unlike real-time tasks scheduled by RTAI. RTAI_FIFO is a service layer that provides uni-directional read/write buffering for tasks. RTAI_SHM provides an efficient way for tasks to exchange large amounts of data through shared memory. For the sake of clarity, we refer to real-time tasks under RTAI as **real-time RTAI tasks**.

LXRT is an advanced feature for RTAI. It allows users to develop real-time tasks using RTAI's API from the Linux user space, as shown in Fig. 1 (b). While in the user space, real-time tasks have the full range of Linux system calls available. When a real-time task over Linux is initialized (by invoking rt_task_init()), a buddy task under RTAI is also created to execute the RTAI function invoked by the soft real-time task running in the Linux user space. The rt_task_wait_period() serves as an example function to illustrate the interactivity between a real-time task and its corresponding buddy task, where rt_task_wait_period() is used to suspend the execution of a real-time task until the next period. When a real-time task over Linux invokes rt_task_wait_period() via a 0xFC software trap, the corresponding buddy task is awakened and becomes ready to execute rt_task_wait_period(). In the invocation, the buddy task delays its resumption time until the next period, and LXRT executes lxrt_suspend() to return CPU control to Linux. We will refer to real-time tasks (supported by RTAI) in the Linux user space as **real-time LXRT tasks** in the following. Note that all tasks under RTAI are threads. For the rest of this paper, we will use the terms "tasks" and "threads" interchangeably when there is no ambiguity.

3. BUDGET-BASED QOS GUARANTEE

3.1 Motivation

The concept of budget-based resource reservation was first proposed by Mercer *et al.* [3, 4, 16] of Carnegie Mellon University. Under budget-based resource reservation, each real-time task is given an execution budget C_i during each specified period P_i, regardless of the arrival of any other higher-priority task. Budget-based resource reservation is considered to be high-level resource allocation, compared to priority-driven resource allocation. The computation power of the system is partitioned among tasks which require QoS support. Near concepts were originally implemented under Windows CE 3.0 (and later versions) as the *thread quantum* and under Unix-like operating systems as the *time quantum*. A task with a specified thread quantum could run until the task uses up the quantum, the task voluntarily gives up the CPU, or a higher-priority task arrives. However, the concept of budget-based resource reservation provides a more intuitive policy that ensures an application with resource allocation. In the rest of this section, we first summarize related RTAI API's and revisions made to provide hard budget guarantees for hard real-time tasks. Our implementation of soft budget guarantees for LXRT tasks and their imprecision issues are then addressed.

3.2 Semantics and Syntax of Budget-Based RTAI API's

RTAI API's include the initialization and manipulation of real-time RTAI tasks and

real-time LXRT tasks. A real-time RTAI task can be initialized by rt_task_init() with the entry point of the task function, a priority, etc. The invocation of rt_task_init() creates a corresponding real-time RTAI task but leaves it in a suspended state. Users must invoke rt_task_make_periodic() to set the starting time and the period of the task. A periodic real-time RTAI task is usually implemented as a loop. At the end of the loop, the real-time RTAI task invokes rt_task_wait_period() to wait for the next period.

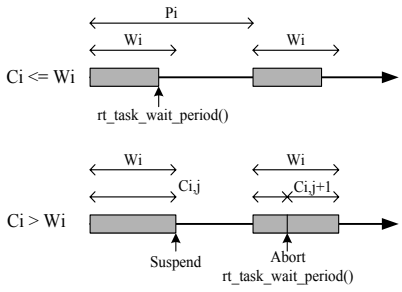
LXRT provides a unique API contact window for acquiring RTAI services. All inline functions in rtai_lxrt.h perform a software interrupt, 0xFC, to request RTAI services, where Linux system calls use the software interrupt 0x80. The interrupt vector call rtai_lxrt_handler() is performed to pass parameters and transfer execution to the corresponding buddy task in the kernel space. A real-time LXRT task can be also initialized by rt_task_init() (which resides in the header file rtai_lxrt.h and differs from the RTAI counterpart) with a unique task name, a priority, etc. When a real-time LXRT task calls rt_task_init(), rt_task_init() generates a buddy task in the kernel space for the real-time LXRT task in order to access RTAI services. Users also need to invoke rt_task_make_periodic() to set the starting time and the period of the real-time LXRT task, implement the task body as a loop, and invoke rt_task_wait_period() in order to wait for the next period.

In order to support budget-based resource reservation for real-time tasks, we propose the following revision of RTAI API's: A real-time RTAI/LXRT task with budget-based resource reservation can be initialized by invoking rt_task_make_periodic_budget(), instead of rt_task_make_periodic(). The format of this new function is exactly the same as the original one, except that an additional parameter for the requested execution budget is provided. A real-time task τ_i can request an execution budget W_i for each of its period P_i. Suppose that the maximum execution time of τ_i is C_i, and $C_i \le W_i$. The execution of τ_i will remains the same without budget reservation because τ_i always invokes rt_task_ wait_period() before it runs out of its budget. It is for the backward compatibility of the original RTAI design. When $C_i \le W_i$, the execution of τ_i might be suspended (before the invocation of rt_task_wait_period()) until the next period because of the exhaustion of the execution budget. The remaining execution of the former period might be delayed to execute in the next period. If that happens (e.g., $C_{i,j} > W_i$, where $C_{i,j}$ denotes the execution time of the task in the j-th period), then the invocation of rt_task_wait_period() (that should happen in the former period) in the next period (i.e., the $(j + 1)$-th period) will be simply ignored, as shown in Fig. 2 (a). The rationale behind this semantics is to let the task gradually catch up the delay, due to overrunning in some periods (that is seen very often in control-loop-based applications).

3.3 An Implementation of Budget-Based Resource Reservation

3.3.1 Hard budget-based resource reservation

Additional members must be included in the process control block of a real-time RTAI task (rt_task_struct). We revise minor parts of the RTAI scheduler (rt_schedule()) and the timer interrupt handler (rt_timer_handler()) to guarantee hard budget-based resource reservation.



(a)

```
struct rt_task_struck {
    …
// appended to provided budget-based resource reservation
#ifdef RMMS
    RTIME assigned_budget;      // assigned execution budget
    RTIME remaining_budget;     // remaining execution budget
    int rttimer_flag;           // flag for timer expiration
    int wakeup_flag;            // flag for budget exhaustion
#ifdef RMS_1
    int if_wait;                // number of delayed invocations
    int force_sig;              // posting a budget-related signal
#endif
#endif
};
```

(b)

Fig. 2. (a) The execution of a real-time task with budget reservation. (b) Additional members of the process data structure for budget-based resource reservation.

In Fig. 2 (b), the additional members assigned_budget and remaining_ budget are for the reserved execution budget and the remaining execution budget of the corresponding task in a period, respectively. The remaining execution budget within a period is modified whenever some special event occurs, such as a change of the task status. rttimer_flag serves as a flag that denotes whether the corresponding real-time task is suspended by means of a timer expiration or a rt_task_wait_period() invocation. wakeup_flag denotes that the corresponding real-time task is suspended because the budget is exhausted. if_wait counts the number of delayed invocations of rt_task_wait_period(), where an invocation is considered to be delayed if it is not invoked in the supposed period because the budget is exhausted. force_sig is set when a budget-related signal is posted to the corresponding LXRT task (see section 3.3.2).

The implementation of budget-based resource reservation in RTAI takes two important issues into consideration: (1) Correct programming of the timer chip (e.g., 8254): Because the resumption and suspension of a real-time RTAI task are both driven by the timer, the budget consumption must be considered when programming of the timer chip. That is, if a task used up its budget before it calls rt_task_wait_period(), then the system must suspend the task execution until the next period, as shown in Fig. 2 (a). (2) The behavior of rt_task_wait_period(): If a task uses up its budget before it calls rt_task_wait_period(), the execution of τ_i might be suspended (before the invocation of rt_task_wait_period()). Execution of the remaining code of the former period might be delayed until the next period. As a result, it is possible to have more than one invocation of rt_task_wait_period() in a period because of delayed invocations. The invocation that should happen in the given period will be simply ignored in the next period.

The RTAI scheduler (rt_schedule()) and timer interrupt handler (rt_timer_handler()) must be revised to guarantee hard budget-based resource reservation. Whenever the timer expires, rt_timer_handler() is invoked. rt_timer_handler() is used to re-calculate the next resume times of the running task and the to-be-awakened task, and to then trigger rescheduling. We propose to revise rt_timer_handler() as follows to take task budgets into consideration.

The value in rt_times.intr_time denotes the next resume time of the running real-time RTAI task. Line 4 derives the resume time based on the remaining budget for the task, where rt_times.tick_time is the current time. If the task will use up its budget

```
1  static void rt_timer_handle (void) {
2      …
3      // calculate the remaining budget
4      temp_time = rt_times.tick_time + new_task → remaining_budget;
5      if (temp_time < rt_times.intr_time) {
6          // assigned_budget is used up
7          new_task → remainig_budget = 0;
8          rt_times.intr_time = temp_time;
9      } else {
10         // assigned_budget is not used up
11         nes_task → remaining_budget –= (rt_times.intr_time – rt_times.tick_time);
12     }
13     …
```

before the next resume time (i.e., rt_times.intr_time), then the code in lines 7 and 8 change the next resume time to the time when the budget will be used up. Otherwise, the remaining budget time at the next resume time is recalculated in line 11. As shown in Fig. 2 (a), a real-time RTAI task might use up its budget before it invokes rt_task_wait_period() (i.e., $C_i > W_i$). Lines 7 and 8 in the above code reflect the need to modify the timer so that the task will be suspended in case of budget overrun. A real-time task might voluntarily give up the CPU by invoking rt_task_wait_period(), which will then call rt_schedule() to dispatch another proper task. The voluntary surrendering of the CPU can occur either when the maximum amount of execution time is not larger than the reserved budget (i.e., $C_i \leq W_i$), or when the de facto execution time is less than the reserved budget, even if $C_i > W_i$. The invocation of rt_task_wait_period() should trigger derivation of the next resume time of the task in rt_schedule().

If a task τ_i used up its budget before it calls rt_task_wait_period(), the execution of τ_i will be suspended (before the invocation of rt_task_wait_period()). As explained in section 3.2, the idea behind rt_task_wait_period() is to ignore any invocation of the function that should happen in some previous period. rt_timer_handler() is also revised as follows to reflect this idea:

3.3.2 Soft budget-based resource reservation

A real-time LXRT task is also initialized by invoking rt_task_init(), and a buddy RTAI task is created by RTAI to execute the RTAI services requested by the real-time LXRT task. Let RT_Task1 be the buddy task of a real-time LXRT task, Task1. When the next period of Task1 arrives, Task1 resumes execution, as shown in Fig. 3. When the timer expires, RTAI schedules RT_Task1. RT_Task1 is suspended right away, because it is a buddy task. CPU execution is transferred to the Linux kernel such that Task1 is scheduled. On the other hand, when Task1 requests any RTAI services through LXRT, such as rt_task_wait_period(), Task1 is suspended, and RT_Task1 resumes execution to invoke the requested RTAI service (i.e., rt_task_ wait_period() in RTAI). The budget information of each real-time LXRT task is maintained in the rt_task_struct of its corresponding buddy task.

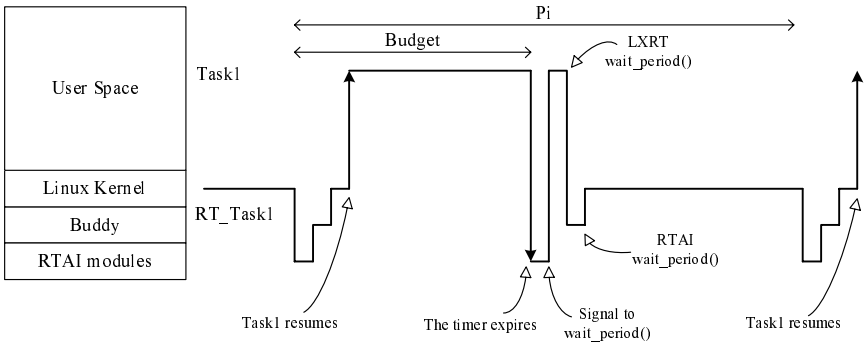


Fig. 3. Scheduling flowchart of the revised LXRT

There are two major challenges when implementing soft budget reservation for real-time LXRT tasks, besides the challenges of correct timer-chip programming and rt_task_wait_period() modification: (1) How can a real-time LXRT task be interrupted when its budget is exhausted and CPU execution be transferred to a proper task? (2) When a higher-priority real-time LXRT task arrives, how can a lower-priority real-time LXRT task and be interrupted and a higher-priority real-time LXRT task dispatched?

We propose to use *signals* to solve the first challenging item in the previous paragraph without any change to the Linux source code. Given a real-time LXRT task, Task1, let RT_Task1 be its buddy RTAI task. When the budget of Task1 is exhausted in the current period, the timer will expire such that a signal of a specified type, e.g., SIGUSR1, will be posted to Task1. The signal handler of the specified type is registered as a function, wait(), that only contains the invocation of rt_task_wait_period(). The signal type for this purpose is referred to as *SIGBUDGET*[3]. The signal posting is done within rt_timer_handler(). Catching the SIGBUDGET signal will result in invocation of rt_task_wait_period() such that Task1 and its buddy task RT_Task1 will be suspended until the next period (for budget replenishing), as shown in Fig. 3.

The second challenge for implementing of budget-based resource reservation for real-time LXRT tasks is triggering rescheduling for higher-priority tasks when they arrive. A different signal number, e.g., SIGUSR2, is used to trigger rescheduling, referred to as *SIGSCHED*. The signal handler of SIGSCHED is registered as a function, preempted(), that only contains the invocation of lxrt_preempted().

rt_timer_handler() should be modified for the purpose of posting SIGBUDGET and SIGSCHED signals. The code in lines 6 to 15 posts of SIGBUDGET signals, because the budget is exhausted. The IF condition in line 4 verifies the need for soft budget-based resource reservation. Line 6 checks the remaining budget of the running real-time LXRT task. If the budget is exhausted, then it is replenished (line 8), and a delay flag is set because of the overrunning of the task. Note that this situation might result in more than one invocation of rt_task_wait_period() in the following period, as discussed in the previous section. We set a reasonable limit on the maximum number of delayed invocations of rt_task_wait_period(), i.e., 30,000. No extra error handling code

[3] Note that new signal numbers could be created in Linux whenever needed (under limited constraints).

```
1  static void rt_timer_handler (void) {
2      …
3      // It is to verify whether soft budget-based resource reservation is needed.
4      If ((rt_wakeup != NULL) && (rt_current → tid == 0) && (rt_wakeup → soft == 1) 0 {
5          …
6          if (((current_rt = current → this_rt_task [0]) != NULL)
7             && (current_rt → tid == wakeup_tid) && (current_rt → remainig_budget == 0)) {
8              current_rt → remaining_budget = current_rt → assigned_budget;
9              current_rt → rttimer_flag = 1;
10             current_rt → if_wait++;
11             if ((current_rt → if_wait) > 30000) current_rt → if_wait = 2;
12             current_rt → force_sig = 1;
13             force_sig (SIGUSR1, current);
14             goto END_rt_timer_handler;
15         }
16         // It is to verify the needs for rescheduling
17         else if (((current_rt → this_rt_task [0]) != NULL)
18           && (current_rt → tid == wakeup_tid) && (current_rt → remaining_budget != 0)) {
19             force_sig (SIGUSR2, current);
20             goto END_rt_timer_handler;
21         }
22         …
```

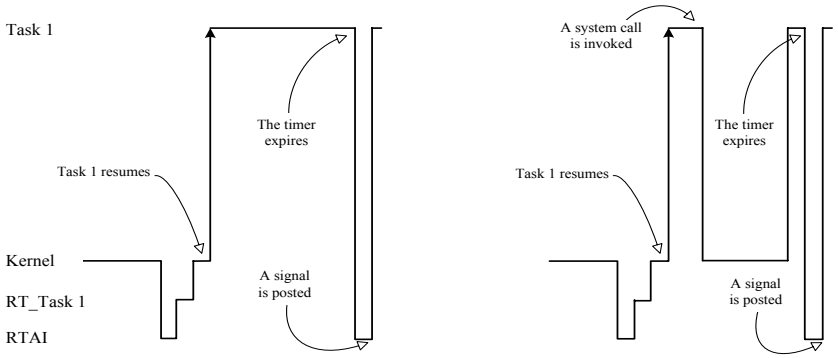
is included in the implementation. A flag for signal posting (for housekeeping purposes in our program) is set (line 12), and a SIGBUDGET signal is posted for the running real-time LXRT task (line 13). The code in lines 17 to 21 posts SIGSCHED signals, because of the need for rescheduling. The IF condition in lines 17 and 18 verifies the need for rescheduling. Note that if the time expires and the budget of the running real-time LXRT task is not exhausted, rescheduling might be needed. A SIGSCHED signal is posted if the condition is satisfied. The goto statement results in a jump to the end of rt_timer_handler().

The following example code is for the implementation of a real-time LXRT task. The registrations of signal handlers for SIGBUDGET (i.e., SIGUSR1) and SIGSCHED (i.e., SIGUSR2) are done in lines 4 and 5. Line 7 initializes of a real-time LXRT task. Line 9 sets up the budget for the task. The loop from line 10 to line 13 is example code for the implementation of a periodic task.

3.3.3 Remarks on the implementation of soft budget-based resource reservation

The implementation of soft budget-based resource reservation is limited in terms of the precision of budget reservation, due to the signal posting/delivery mechanism in Linux. When a SIGBUDGET or SIGSCHED signal is posted to the running real-time LXRT task, Task1, the task might be running in the user space or be suspended when a system call is run. If the real-time LXRT task is running in the user space when the time expires, the timer interrupt will trigger the execution of rt_timer_handler() and post a proper signal. As a result, when the timer handler returns, the signal will be delivered to

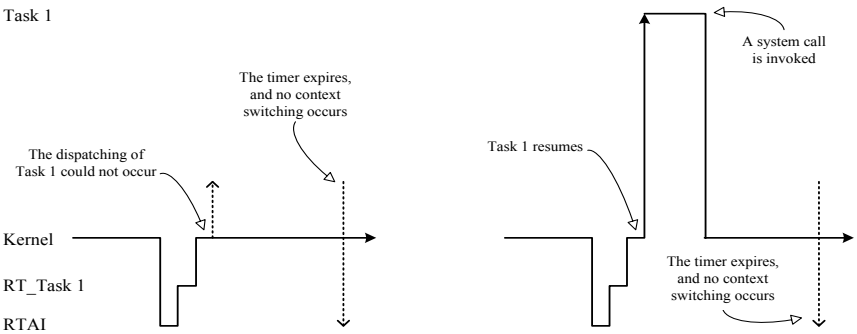
the real-time LXRT task, as shown in Fig. 4. The corresponding signal handler will invoke rt_task_wait_period() or lxrt_preempted(). In this situation, budget reservation will be pretty accurate.



(a) No system call is involved. (b) The system call is done.

Fig. 4. The time expires when the real-time LXRT task is running in the user space.

The above imprecision of budget-based resource reservation mainly occurs when the Linux kernel is executing while the time expires. When RTAI dispatches the buddy RTAI task of a real-time LXRT task, the Linux kernel might be running in the kernel space. Although the timer interrupts the Linux kernel, the Linux kernel still stays in the kernel space until the system call service finishes, as shown in Fig. 5 (a). As a result, execution of the dispatched real-time LXRT task is delayed. This may result in miscalculation of the remaining budget of the dispatched real-time LXRT task. If rescheduling is needed or the budget of a running real-time LXRT task is exhausted when the Linux kernel is servicing a system call invoked by the task, then another delay may occur, because neither the posted SIGSCHED signal nor the posted SIGBUDGET signal can be delivered to the running task on time, as shown in Fig. 5 (b).



(a) Delay on the resumption of the scheduled LXRT task. (b) Dalay due to the service of a system call.

Fig. 5. The Linux kernel is executing while the timer expires.

Either of the two cases shown in Fig. 5 will result in imprecise for budget-based resource reservation. One possible solution is to compensate for the loss of the dispatched real-time LXRT task. The technical problem is determining how much to compensate for the loss! We take a novel approach, where the real-time LXRT task that suffers from budget loss is set to run freely until it invokes rt_task_wait_period(). This approach is simple but might not make up for the budget loss. Other higher-priority real-time LXRT tasks might also have a chance to preempt the task that suffers from budget loss. A better approach would be to measure the loss in the timer handler and then compensate for this loss.

4. SYSTEM OVERHEADS AND BUDGET PRECISION

The experiments consisted of two parts: The first part focused on the system overheads of the implementation for real-time RTAI tasks. The second part focused on the system overheads and budget precision of the implementation for real-time LXRT tasks. We do not include the experimental results for budget precision for real-time RTAI tasks, because they were very accurate and only depended on the dispatching latency (including the interrupt latency) of RTAI. The experiments were conducted over RedHat Linux 7.0 (Linux kernel 2.4.0-test10) patched with RTAI 24.1.2. We then patched Linux-RTAI with the proposed code for budget-based resource reservation. In addition, Linux Trace Toolkit (LTT) version 0.9.4 was adopted as the trace tool to monitor the whole scheduling behavior. Different x86 hardware platforms were tested as shown in Table 1.

Table 1. Testing hardware platforms.

CPU	Clock Frequency	L1		L2 Cache
		I-Cache	D-Cache	
PIII 500	497.55 MHz	16 KB	16 KB	256 KB
PIII Celeon 400	399.06 MHz	16 KB	16 KB	128 KB
PII 266	266.67 MHz	16 KB	16 KB	512 KB

4.1 Real-Time RTAI Tasks

In order to evaluate the system overheads of the implementation, we measured the actual budget received by a real-time RTAI task, referred to as the *received budget*. The assigned budget of a real-time RTAI task is the amount specified by users for resource reservation, referred to as the *assigned budget*. As a result, the received budget of a real-time RTAI task is the assigned budget minus the system overheads (if no compensation mechanism is adopted). Let the number of timer expirations during each servicing of the assigned budget be N, which was proportional to the extra overheads and could be changed among periods: $ReceivedBudget = AssignedBudget - N \times SystemOverhead$.

The task set in these experiments consisted of two tasks $\{\tau_1, \tau_2\}$, where the period and the assigned budget of τ_1 were 2 *ms* and 1 *ms*, respectively, and the period and the

assigned budget of τ_2, were 5 *ms* and 1 *ms*, respectively, These periods and budgets were selected only because they seemed reasonable and could provide a reference for the measurement of the system overheads.

The experimental results were shown in Table 2. The simulation time was 500 *ms*. The average overhead were derived by dividing the total system overhead by the number of timer expirations during the simulation time. The maximum overhead were the maximum observed system overhead for a timer expiration during the simulation time. It was found that the system overheads were very limited. For example, even on the PII 266 system, the maximum overhead was only 0.01 *ms*, and the average overhead was less than 0.007 *ms*.

Table 2. System overheads of the implementation for real-time RTAI tasks.

Platform	Average Overhead	Maximum Overhead
PIII 500	0.00578 *ms*	0.007 *ms*
PIII Celeon 400	0.00688 *ms*	0.008 *ms*
PII 266	0.00638 *ms*	0.010 *ms*

4.2 Real-Time LXRT Tasks

The experiments in the second part were conducted to evaluate the system overheads and budget precision of the implementation for real-time LXRT tasks. Budget precision was measured based mainly on the amount of received budget, rather than the assigned budget. The system overheads reported in this section included the original RTAI overheads in scheduling/interrupt handling and extra overheads resulting from execution of the additional code discussed in section 3. In reality, the system overheads were difficult to measure, so the guarantee for the received budget could merely be considered soft. The relationship between an assigned budget and the corresponding received budget was modified as follows: $ReceivedBudget = AssignedBudget - N \times SystemOverhead - KernelDelays$. KernelDelays denotes the dispatching delay due to the processing of system services, as shown in Fig. 5. The three x86 hardware platforms were tested again for the experiments on the implementation for real-time LXRT tasks.

4.2.1 System overheads

The same task set τ_1, τ_2 adopted in section 4.1 was adopted in the simulation to facilitate a comparison with the results reported in section 4.1, except that τ_1 and τ_2 were now real-time LXRT tasks. Table 3 summarizes the system overheads measured over the different x86 platforms. The simulation time was 500 *ms*. The average overhead were derived by dividing the total system overhead by the number of timer expirations during the simulation time. The maximum overhead were the maximum observed system overhead for a timer expiration during the simulation time. All the results were gathered over LTT. The system overheads were higher than the corresponding results in Table 3. The system overheads were still limited however. For example, even on the PII 266 system,

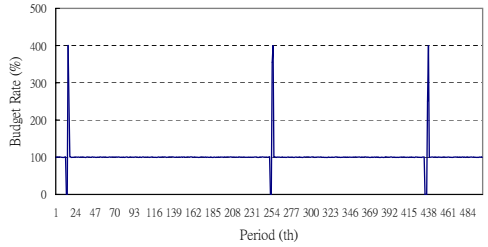
Table 3. System overheads of the implementation for real-time LXRT tasks.

Platform	Average Overhead	Maximum Overhead
PIII 500	0.01982 *ms*	0.035 *ms*
PIII Celeon 400	0.02708 *ms*	0.056 *ms*
PII 266	0.03469 *ms*	0.096 *ms*

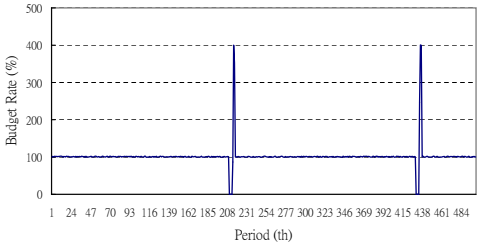
the maximum overhead was only 0.096 *ms*, and the average overhead was less than 0.03469 *ms*. This was mainly because of the extra work needed for the implementation, such as signal posting/delivery and context switching between the user space and kernel space.

4.2.2 Budget precision

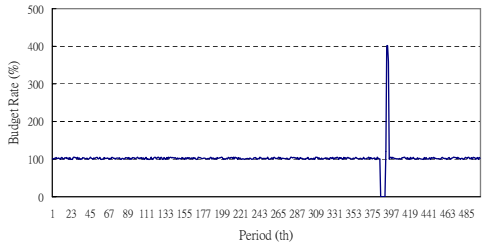
Since the real-time LXRT tasks were Linux tasks, the experiments on budget precision for the implementation were complicated by the inclusion of more tasks with different characteristics, such as the playing of MPEG streams. These experiments consisted of four real-time LXRT tasks $\{\tau_1, \tau_2, \tau_3, \tau_4\}$. τ_1, τ_2, and τ_3 were simple computation-intensive real-time LXRT tasks with the same CPU utilization, and τ_4 was an MPEG player. The period and the assigned budget of τ_1 were 4 *ms* and 1 *ms*, respectively. The period and assigned budget of τ_2 were 2 *ms* and 0.5 *ms*, respectively. The period and assigned budget of τ_3 were 1 *ms* and 0.25 *ms*, respectively. τ_4 was an MPEG player with a period of 20 *ms* and a budget of 5 *ms*. τ_4 was a real-time LXRT task that read from the disk for MPEG streams and output the decoded video through the X server.



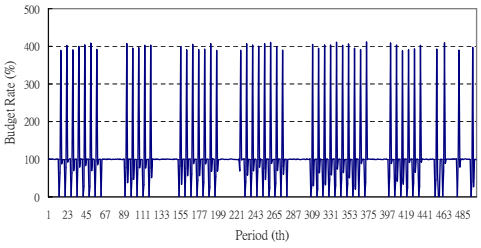
(a) τ_1 (Period, Budget) = (4 *ms*, 1 *ms*).



(b) τ_2 (Period, Budget) = (2 *ms*, 0.5 *ms*).



(c) τ_3 (Period, Budget) = (1 *ms*, 0.25 *ms*).



(d) MPEG τ_4 (Period, Budget) = (20 *ms*, 5 *ms*).

Fig. 6. Budget precision during 500 periods.

The precision of soft budget-based resource reservation was measured based on the ratio of the received budget to the assigned budget, referred to as the *budget rate*:

$$BudgetRate(\%) = \frac{ReceiveBudget}{AssignedBudget} \times 100\%.$$

Figs. 6 (a), (b), and (c) show the precision of soft budget-based resource reservation results for τ_1, τ_2, and τ_3, respectively. The X-axis denotes the number of periods in the experiment, and the Y-axis denotes the budget precision in terms of the budget rate. It is shown that the budget precision was 100% most of the time. During the entire experiment, no more than 3 violations were observed over 500 periods. When the assigned budget was small, violation was more likely, because the assigned budget was large, compared to the system overheads, as shown in Table 3. Fig. 6 (d) shows the precision of soft budget-based resource reservation for the MPEG player. Many more violations occurred, because the MPEG player made a lot of system calls, and synchronization existed between the MPEG player and the *X* server. In about 10% of the periods, the MPEG player received a budget that was different from the assigned budget. However, the quality of MPEG movie playing still seemed very good.

5. CONCLUSIONS

The purpose of this paper has been to explore issues related to budget-based resource reservation for real-time tasks over Linux and propose an implementation. We have extended RTAI API's to provide both hard budget guarantees for hard real-time tasks under RTAI and soft budget guarantees for LXRT tasks in Linux user space. Backward compatibility is maintained with the original RTAI (and LXRT) design. Modifications of RTAI are limited to few procedures, such as the timer interrupt handler, RTAI scheduler, and rt_task_wait_period(). A series of experiments are conducted to evaluate the overheads and budget precision of the proposed implementation. It was shown that the system overheads were very limited for resource reservation for real- time RTAI tasks and real-time LXRT tasks. The precision of budget-based resource reservation was also very good, even for a complicated workload of an MPEG player.

In future research, we shall further extend our implementation to multi-threading processes for sharing of a single budget. We shall also explore synchronization issues related to cooperating processes, especially when a budget is reserved for each process. A joint management scheme for multiple resources, such CPUs and devices, will also be explored.

REFERENCES

1. M. Jones, D. Rosu, and M. Rosu, "CPU reservation and time constraints: efficient, predictable scheduling of independent activities," *ACM Symposium on Operating Systems Principles*, 1997, pp. 198-211.
2. T. W. Kuo, G. H. Huang, and S. K. Ni, "A user-level computing power regulator for

soft real-time applications on commercial operating systems," *Journal of the Chinese Institute of Electrical Engineering*, Vol. 6, 1999, pp. 13-25.
3. C. W. Mercer and R. Rajkumar, "An interactive interface and RT-mach support for monitoring and controlling resource management," *IEEE Real-Time Technology and Applications Symposium*, 1995, pp. 134-139.
4. C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: an abstraction of managing processor usage," in *Proceedings of 4th Workshop on Workstation Operating Systems* (*WWOS-IV*), 1993, pp. 129-134.
5. S. Wang, K. J. Lin, and Y. Wang, "Hierarchical budget management in the RED-Linux scheduling framework," in *Proceedings of 14th EUROMICRO Conference on Real- Time Systems*, 2002, pp. 76-83.
6. Y. Wang and K. J. Lin, "Enhancing the real-time capability of the Linux kernel," in *Proceedings of 5th Real-Time Computing Systems and Applications Symposium*, 1998, pp. 11-20.
7. Z. Deng and J. W. S. Liu, "Scheduling real-time applications in an open environment," *IEEE Real-Time Systems Symposium*, 1997, pp. 308-319.
8. M. Spuri, G. Buttazzo, and F. Sensini, "Scheduling aperiodic tasks in dynamic scheduling environment," *IEEE Real-Time Systems Symposium*, 1995, pp. 179-210.
9. I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," *IEEE Real-Time Systems Symposium*, 1996, pp. 288-299.
10. H. Neugass, G. Espin, H. Nunoe, R. Thomas, and D. Wilner, "VxWorks: an interactive development environment and real-time kernel for GMicro," in *Proceedings of 8th TRON Project Symposium*, 1991, pp. 196-207.
11. L. Thompson, "Using pSoS+ for embedded real-time computing," in *Proceedings of 35th IEEE Computer Society International Conference*, 1990, pp. 282-288.
12. P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, S. Hughes, and K. Yaghmour, "DIAPM-RTAI position paper," in *Proceedings of Real-Time Linux Workshop*, 2000.
13. V. Yodaiken, "The RTLinux manifesto," in *Proceedings of 5th Linux Expo*, 1999, pp. 187-197.
14. Y. Wang and K. J. Lin, "Implementing a general purpose real-time scheduling framework in the RED-Linux real-time kernel," in *Proceedings of IEEE Real-Time Systems Symposium*, 1999, pp. 246-255.
15. J. A. O'Keefe, "Venturcom RTX enabling Microsoft windows XP and windows XP embedded for hard real-time," Venturcom, Inc., Tech. Rep., 2002.
16. C. W. Mercer, R. Rajkumar, and J. Zelenka, "Temporal protection in real-time operating systems," in *Proceedings of 11th IEEE Workshop on Real-Time Operating Systems and Software*, 1994, pp. 79-83.
17. L. P. Chang, T. W. Kuo, and S. W. Lo, "A dynamic-voltage-adjustment mechanism in reducing the power consumption of flash memory for portable devices," in *Proceedings of IEEE International Conference on Consumer Electronics*, 2001, pp. 218-219.
18. H. M. Chen, S. Y. Zhuo, C. Y. Huang, and T. W. Kuo, "An USB-based surveillance system over wireless network," in *Proceedings of 7th International Conference on Distributed Multimedia Systems*, 2001, pp. 510-513.

19. M. L. Hsu, W. R. Yang, Y. T. Kao, G. H. Huang, and T. W. Kuo, "Providing real-time access control to remote resources," in *Proceedings of 3rd Workshop on Real-Time and Media Systems* (*RAMS '97*), 1997, pp. 137-143.
20. G. H. Huang, S. K. Ni, and T. W. Kuo, "The design and implementation of the CPU power regulator for multimedia operating systems," in *Proceedings of 17th IEEE Real-Time Systems Symposium* (*RTSS '96*), 1996, pp. 27-30.
21. T. W. Kuo and C. H. Li, "A fixed-priority-driven open environment for real-time applications," in *Proceedings of 20th IEEE Real-Time Systems Symposium*, 1999, pp. 256-267.
22. B. Adelberg, H. Garcia-Molina, and B. Kao, "Emulating soft real-time scheduling using traditional operating systems schedulers," in *Proceedings of 15th IEEE Real-Time Systems Symposium*, 1994, pp. 292-298.
23. S. Childs and D. Ingram, "The Linux-SRT integrated multimedia operating systems: Bring QoS to the desktop," in *Proceedings of IEEE Real-Time Technology and Applications Symposium*, 2001, pp. 135-140.
24. L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A measurement-based analysis of the real-time performance of Linux," in *Proceedings of 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002, pp. 133-142.
25. DIAPM RTAI Programming Guide 1.0, Lineo, Inc., 2000.

Nei-Chiung Perng (彭念劬) received his Bachelor and Master degrees in Computer and Information Science from National Chao Tung University in 1999 and 2001, respectively. He is currently pursuing the Ph.D. degree in Computer Science and Information Engineering at National Taiwan University. His research interests include real-time systems and scheduling algorithms.

Chin-Shuang Liu (劉進雙) is currently an assistant manager in the Department of Mobile Product Development of FIH. He received his master degree in the Department of Computer and Information Science, National Taiwan University in 2002. His working experiences include designs of system architecture and firmware on mobile devices for end users.

Tei-Wei Kuo (郭大維) received the B.S.E. degree in Computer Science and Information Engineering from National Taiwan University in Taipei, Taiwan, in 1986. He received the M.S. and Ph.D. degrees in Computer Sciences from the University of Texas at Austin in 1990 and 1994, respectively. He is currently a Professor and the Chairman at the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, ROC. His research interests include embedded systems, real-time operating systems, and real-time database systems.