An Implementation of Budget-Based Resource Reservation for Real-Time Linux*

C.S. Liu, N.C. Perng, and T.W. Kuo

Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan 106, ROC

Abstract. 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. Different from the past work, we focus on extending RTAI API's such that programmers could specify a computation budget for each task, and the backward compatibility is maintained. Modifications on RTAI are limited to few procedures without any change to Linux kernel. The feasibility of the proposed implementation is demonstrated by a system over Linux 2.4.0-test10 and RTAI 24.1.2 on PII and PIII platforms.

1 Introduction

Various levels of real-time support are now provided in many modern 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 [1,2,3] and rate-based scheduling [4,5,6,7]. The concept of budget-based reservation, that is considered as an important approach for applications' QoS support, was first proposed by Mercer, et al. [8]. 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 [3]. There were also many other research and implementation results on the QoS support for real-time applications. In particular, Adelberg, et al. [9] presented a real-time emulation

* This research was supported in part by the National Science Council under grant NSC91-2213-E-002-104

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 226–233, 2004.

program to build soft real-time scheduling on the top of UNIX. Childs and Ingram [10] 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 [11]. 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. Swaminathan, et al. explored energy consumption issues in real-time task scheduling over RTLinux [12].

The purpose of this paper is to explore budget-based resource reservation for real-time tasks which run over Linux and propose its implementation. We first extend Real-time Application Interface (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 Linux tasks with LXRT over that for hard real-time tasks under RTAI. Backward compatibility is maintained for the original RTAI (and LXRT) design. The software framework and patch for the proposed implementation is presented, and 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()`. The feasibility of the proposed implementation is demonstrated over Linux 2.4.0-test10 and RTAI 24.1.2 on PII and PIII platforms.

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. Section 4 is the conclusion.

2 RTAI

RTAI is one the most popular real-time patches to let Linux provide deterministic and preemptive performance for hard real-time tasks [13]. LXRT is an advanced feature for RTAI. It allows users to develop real-time tasks using RTAI's API from the Linux user space. While in the user space, real-time tasks have the full range of Linux system calls available. Besides, LXRT also provides the same set of RTAI API calls available for RTAI applications in the Linux user space. 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 to suspend the execution of a real-time task until the next period. When a real-time tasks over Linux invokes `rt_task_wait_period()` via 0xFC software trap, the corresponding buddy task is waken up and becomes ready to execute

[1] The approach is very different from that of this paper. We intend to propose an RTAI-based implementation to deliver budget-based reservations to hard and soft real-time applications.

`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 back to Linux.

We refer real-time tasks (supported by RTAI) in the Linux user space as *real-time LXRT tasks* for future discussions, distinct from real-time tasks under RTAI (referred to as *real-time RTAI tasks*). Note that all tasks under RTAI are threads. For the rest of this paper, we will use terms tasks and threads interchangeably when there is no ambiguity.

3 Budget-Based QoS Guarantee

The concept of budget-based resource reservation is considered as a high-level resourse allocation concept, compared to priority-driven resource allocation. Each real-time task τ_i is given an *execution budget* W_i during each specified period P_i. The computation power of the system is partitioned among tasks with QoS requirements. The concept of budget-based resource reservation could provide an intuitive policy that ensures an application with resource allocation can always run up to its execution budget. Although some researchers have proposed excellent implementation work and designs for budget-based resource reservations in Linux (or other operating systems) or emulation of real-time support over Linux, little work is done on the exploring of the implementations for resource reservations over both Linux and RTAI (that is considered under Linux), and even for real-time LXRT tasks. Note that real-time LXRT tasks run over Linux, but they also invoke RTAI services.

3.1 Semantics and Syntax of Budget-Based RTAI APIs

A real-time RTAI task can be initialized by the RTAI API `rt_task_init()` with the entry point of the task function, a priority, etc. The invocation of `rt_task_init()` creates the 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.

Non-root users develop real-time tasks in the Linux user space with LXRT. All inline functions in `rtai_lxrt.h` do 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 to pass parameters and to transfer the 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. This API generates a buddy task in the kernel space for the real-time LXRT task to access RTAI services. Users then do the same work as they program a real-time RTAI task.

In order to support budget reservation, we revise and propose new RTAI API's as follows: A real-time RTAI/LXRT task with budget-based resource

reservation could be initialized by invoking `rt_task_make_periodic_budget()`, instead of the original `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. Under this model, 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 $C_i \leq W_i$. 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 $C_i > W_i$, 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. 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).

3.2 An Implementation for Budget-Based Resource Reservation

Hard Budget-Based Resource Reservation. Additional attributes are now included in the process control block of real-time RTAI tasks (`rt_task_struct`). We revise some minor parts of the RTAI scheduler (`rt_schedule()`) and the timer interrupt handler (`rt_timer_handler()`) to guarantee hard budget-based resource reservation.

```
struct rt_task_struct {
   ...
   // appended to provided budget-based resource reservation
   RTIME assigned_budget;
   RTIME remaining_budget;
   int rttimer_flag;
   int wakeup_flag;
   int if_wait;
   int force_sig;
};
```

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 the changes of the task status. `rttimer_flag` serves as a flag to denote whether the corresponding real-time task is suspended by a timer expiration or a `rt_task_wait_period()` invocation. `wakeup_flag` is to denote that the corresponding real-time task is suspended because of budget exhaustion. `if_wait` is used to count the number of delayed invocations of `rt_task_wait_period()`, where an invocation is considered delayed if it is not invoked in the supposed period because of bud-

get exhaustion. `force_sig` is set when a budget-related signal is posted to the corresponding LXRT task.

The implementation of budget-based resource reservation in RTAI must consider two important issues: (1) The correct programming of the timer chip (e.g., 8254). (2) The behavior of `rt_task_wait_period()`. Besides, the RTAI scheduler (`rt_schedule()`) and the timer interrupt handler (`rt_timer_handler()`) must also 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-awaken task and then trigger rescheduling. We propose to revise `rt_timer_handler()` as follows to include the considerations of task budgets.

```
1  static void rt_timer_handler(void) {
2    ...
3    // calculate the remaining budget
4    if (new_task->tid != 0) {
5      temp_time = rt_times.tick_time + new_task->remaining_budget;
6      if (temp_time < rt_times.intr_time) {
7        // assigned_budget is used up
8        new_task->remaining_budget = 0;
9        rt_times.intr_time = temp_time;
10     } else {
11       // assigned_budget is not used up
12       new_task->remaining_budget -= (rt_times.intr_time-rt_times.tick_time);
13     }
14   }
15   ...
16 }
```

The value in `rt_times.intr_time` denotes the next resume time of the running real-time RTAI task. Line 5 derives the resume time based on the remaining budget of the task, where `rt_times.tick_time` is the current time. If the task uses up its budget before the next resume time (i.e., `rt_times.intr_time`), then code in Lines 8 and 9 should modify the next resume time to the time when budget is used up. Otherwise, the remaining budget time at the next resume time is recalculated in Line 12.

As explained in previous paragraphs, the semantics of `rt_task_wait_period()` is to ignore the invocation of the function that should happen in some former periods. `rt_timer_handler()` is also revised to reflect such semantics: If there is no budget remained, and the invocation is for hard budget-based resource reservation, then the remaining budget and the next ready time (i.e., the resume time) of the running real-time RTAI task must be reset.

Soft Budget-Based Resource Reservation. Let a real-time LXRT task $Task1$ be initialized by invoking `rt_task_init()`, and a corresponding buddy RTAI task RT_Task1 is created. When the next period of $Task1$ arrives, $Task1$ resumes through the following sequence, as shown in Fig. 1: When the timer expires, let RTAI schedule RT_Task1. RT_Task1 is suspended right away be-

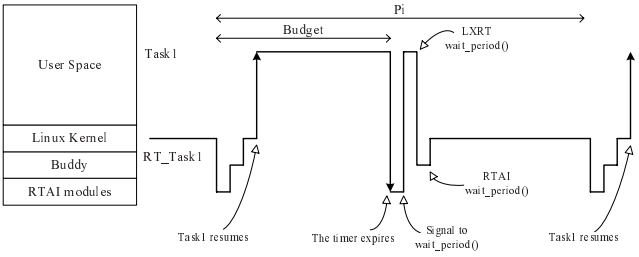

Fig. 1. Scheduling flowchart of the revised LXRT

cause it is a buddy task. The CPU execution is then 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()` (in the user space), $Task1$ is suspended, and RT_Task1 resumes its 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.

There are two major challenges in the implementation of soft budget-based resource reservation for real-time LXRT tasks: (1) How to interrupt a real-time LXRT task when its budget is exhausted and to transfer the CPU execution right to a proper task. (2) When a higher-priority real-time LXRT task arrives, how to interrupt a lower-priority real-time LXRT task and dispatch the higher-priority real-time LXRT task.

We propose to use *signals* to resolve the first challenging item. Note that we wish to restrict modifications on RTAI without any change to the Linux source code. 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, is 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 such a purpose is referred to as *SIGBUDGET*[2]. The signal posting is done within `rt_timer_handler()`. The catching of the SIGBUDGET signal will result in the invocation of `rt_task_wait_period()` such that $Task1$ and its buddy task RT_Task1 are suspended until the next period (for budget replenishing), as shown in Fig. 2.

The implementation for second challenge item is also based on the signal posting/delivery mechanism. A different signal number, e.g., SIGUSR2, is used for the triggering purpose of rescheduling. The signal type for such a purpose is referred to as *SIGSCHED*. The signal handler of SIGSCHED is registered as a function `preempted()`, that only contains the invocation of `lxrt_preempted()`. Consider two real-time LXRT tasks $Task1$ and $Task2$, where RT_Task1 and RT_Task2 are the corresponding buddy RTAI tasks. Suppose that the priority of $Task1$ is lower than that of $Task2$. We must point out that the arrival of any real-time RTAI task will result in the expiration of the timer because

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

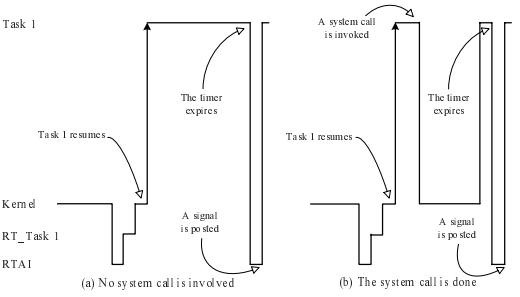

Fig. 2. The timer expires when the running real-time LXRT task is in the user space.

of the setup of the resume time of the task (initially, aperiodically, or periodically). When $Task2$ arrives (while $Task1$ is executing), the timer will expire, and `rt_timer_handler()` is executed to post a signal to the running task, i.e., $Task1$. The SIGSCHED signal is delivered to `lxrt_preempted()` (similar to the signal delivery in Fig. 2). As a result, `lxrt_preempted()` is executed such that $Task1$ is suspended, and the CPU execution right is transferred to RTAI. RTAI realizes the arrival of $Task2$ and dispatches RT_Task2, instead of RT_Task1, because the priority of RT_Task1 is lower than that of RT_Task2, due to the priority order of $Task1$ and $Task2$. The dispatching of RT_Task2 results in the transferring of the CPU execution right to $Task2$ through LXRT.

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

```
1 void wait(){ rt_task_wait_period(); }
2 void preempted(){ lxrt_preempted(); }
3 void main(void) {
4   signal(SIGUSR1,wait);
5   signal(SIGUSR2,preempted);
6   //initialization codes
7   rt_task_init();
8   //use the system call to gain the budget
9   rt_task_make_periodic_budget(srt_task, now + period, period, budget);
10  while(1) {
11    //computation codes that needs budget service.
12    rt_task_wait_period();
13  }
14 }
```

4 Conclusion

We extend RTAI API's to provide hard budget guarantees to hard real-time tasks under RTAI and soft budget guarantees for LXRT tasks over that for real-

time RTAI tasks. Backward compatibility is maintained for the original RTAI (and LXRT) design. We try to minimize the modifications on RTAI without any change to the Linux source code. The feasibility of the proposed implementation is demonstrated over Linux 2.4.0-test10 and RTAI 24.1.2 on PII and PIII platforms. For the future research, we shall further extend the implementation work to multi-threading processes for the sharing of a single budget. We shall also explore the synchronization issues for cooperating processes, especially when a budget is reserved for each of them. A joint management scheme for multiple resources such CPU and devices will also be explored.

References

1. Jones, M., Rosu, D., Rosu, M.: Cpu reservation and time constraints: Efficient, predictable scheduling of independent activities. ACM Symposium on Operating Systems Principles (1997) 198–211
2. Kuo, T.W., Huang, G.H., Ni, S.K.: A user-level computing power regulator for soft real-time applications on commercial operating systems. Journal of the Chinese Institute of Electrical Engineering **6** (1999) 13–25
3. Wang, S., Lin, K.J., Wang, Y.: Hierarchical budget management in the red-linux scheduling framework. 14th Euromicro Conference on Real-Time Systems (2002)
4. Deng, Z., Liu, J.W.S.: Scheduling real-time applications in an open environment. IEEE Real-Time Systems Symposium (1997)
5. Spuri, M., Buttazzo, G., Sensini: Scheduling aperiodic tasks in dynamic scheduling environment. IEEE Real-Time Systems Symposium (1995)
6. Stoica, I., Abdel-Wahab, H., Jeffay, K., Baruah, S., Gehrke, J., Plaxton, C.: A proportional share resource allocation algorithm for real-time, time-shared systems. IEEE Real-Time Systems Symposium (1996) 288–299
7. Waldspurger, C.: ottery and stride scheduling: Flexible proportional-share resource management. Technical report, Ph.D. Thesis, Technical Report, MIT/LCS/TR-667, Laboratory for CS, MIT (1995)
8. Mercer, C.W., Savage, S., Tokuda, H.: Processor capacity reserves: An abstraction of managing processor usage. In Proceedings of the Fourth Workshop on Workstation Operating Systems (WWOS-IV) (1993)
9. Adelberg, B., Garcia-Molina, H., B.Kao: Emulating soft real-time scheduling using traditional operating systems schedulers. IEEE 15th Real-Time Systems Symposium (1994) 292–298
10. Childs, S., Ingram, D.: The linux-srt integrated multimedia operating systems: Bring qos to the desktop. IEEE Real-Time Technology and Applications Symposium, Taipei, Taiwan, ROC (2001) 135–140
11. Abeni, L., Goel, A., Krasic, C., Snow, J., Walpole, J.: A measurement-based analysis of the real-time performance of linux. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (2002)
12. Swaminathan, V., Schweizer, C.B., Chakrabarty, K., Patel, A.A.: Experiences in implementing an energy-driven task scheduler in rt-linux. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (2002)
13. Cloutier, P., Mantegazza, P., Papacharalambous, S., Soanes, I., Hughes, S., Yaghmour, K.: Diapm-rtai position paper. Real Time Operating Systems Workshop (2000)