Transparent and Selective Real-time Interrupt Services for Performance Improvement

Jinkyu Jeong[1], Euiseong Seo[1], Dongsung Kim[1], Jin-Soo Kim[1], Joonwon Lee[1], Yung-Joon Jung[2], Donghwan Kim[2], and Kanghee Kim[3]

[1] Dept. of CS, Korea Advanced Institute of Science and Technology
[2] Electronics and Telecommunications Research Institute
[3] Samsung Electronics Co.
{jinkyu,ses,dskim}@calab.kaist.ac.kr,{jinsoo,joon}@cs.kaist.ac.kr
{jjing,dhkim76}@etri.re.kr,kang.hee.kim@samsung.com

Abstract. The popularity of mobile and multimedia applications made real-time support a mandatory feature for embedded operating systems. However, the current situation is that the overall performance is significantly degraded due to the real-time support. This paper suggests a novel scheme to minimize the performance degradation in embedded operating systems with real-time support. Especially, we propose transparent and selective real-time interrupt services which transparently monitor the system and postpone interrupt handling that are not relevant to real-time tasks. The proposed scheme was implemented on the Linux 2.6 kernel and the experimental results show that our scheme improves the throughput by up to 86% for Hackbench benchmark while providing almost the same scheduling latency compared to the previous work.

Key words: Real-time, Scheduling algorithm, Interrupt handling, Embedded operating systems, Latency, Throughput

1 Introduction

Due to the digital convergence phenomenon [1], consumer electronics devices, such as cell phones, PDAs (Personal Digital Assistants), and PMPs (Portable Media Players), run many sophisticated applications beyond their original purposes. For example, a typical cell phone is equipped not only with the phone tasks for calling and SMS (Short Message Service), but also with PIMS (Personal Information Management System), still pictures management system, and simple games. The number of these extra applications, as well as the size and the complexity of the individual application, will continue to grow rapidly in the near future.

Pure real-time operating systems are not adequate for those consumer electronics devices because of its limited functionality and the lack of generality. As a result, more general embedded operating systems, such as Windows CE and Embedded Linux, are becoming widely used in the area of portable embedded systems.

Many tasks in portable embedded systems are time sensitive [2] or require prompt response to external stimuli. Notable examples include call processing tasks in PDAs or video streaming tasks in PMPs. To make these real-time tasks run harmoniously with other normal tasks, a certain level of real-time support is essential in embedded operating systems. In spite of this requirement, many embedded operating systems which are rooted in general-purpose operating systems do not fully support the real-time constraint.

Recently, Ingo Molnar has proposed a Linux kernel patch called *Complete Preemption* [3] for the improved real-time support in embedded devices. Although Complete Preemption is quite effective in improving the responsiveness of the system [4], the problem is that the system throughput is notably degraded due to the real-time support. We find that the excessive context switching between tasks to provide prompt response to real-time tasks is the main source of the performance degradation.

To resolve this problem, we propose a novel scheme to minimize context switching without sacrificing the responsiveness of the system. The proposed scheme suppresses the preemption by normal tasks so that only the interrupts associated with real-time tasks are rapidly serviced. The interrupts associated with real-time tasks are transparently identified by the system, thus requiring no manual intervention. The proposed scheme was implemented in the Linux kernel 2.6 and evaluated using various benchmarks. Our result shows that the proposed scheme improves the throughput up to 86% for Hackbench benchmark on VIA C3 embedded board.

The rest of the paper is organized as follows. The following section reviews the previous work for real-time support. Section 3 explains Complete Preemption in detail and analyzes the source of performance degradation. Section 4 discusses the proposed scheduling policy of real-time and normal tasks. Section 5 shows the evaluation results compared to the existing solutions in the aspects of scheduling latency and throughput. The final section summarizes our work and concludes the paper.

2 Related work

Although widely used in embedded systems, Linux is hardly classified into real-time operating system (RTOS) since it does not support full preemption and priority inheritance. Two approaches, sub-kernel and preemptible kernel, are proposed to make Linux support real-time applications.

RTLinux [5] from FSMLabs and RTAI [6] are the representative examples of the sub-kernel approach [7]. The kernel is divided into core-kernel and sub-kernel. When some real-time tasks exist, the core-kernel executes those real-time tasks. Otherwise, the control is transferred to the sub-kernel and normal tasks are executed. In the sub-kernel structure, the scheduling latency of real-time tasks becomes lower than tens of microseconds. However, the sub-kernel approach has a disadvantage that only normal tasks can fully exploit the features provided by the Linux kernel.

A preemptible kernel [8] denotes the kernel which can be preempted either at certain preemption points or everywhere inside the kernel. For example, RED Linux [9] inserts preemption points in the kernel, while Timesys Linux/RT [10] supports full kernel preemption. The Linux kernel 2.6 also supports kernel preemption originally developed by the preemptible kernel project [11]. In the preemptible Linux kernel, the kernel can be preemptible if it is not in critical sections, which enhances the responsiveness of real-time tasks. Unlike the sub-kernel approach, real-time tasks can make full use of kernel features. The scheduling latency, however, becomes more or less unstable.

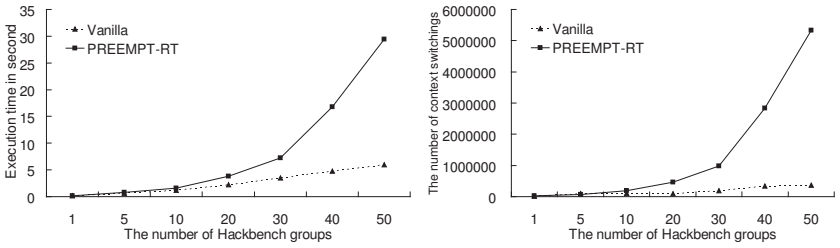
Ingo Molnar's Complete Preemption [3, 4] makes preemption possible even when the kernel is in critical sections using mutex-based spin locks. ISRs (Interrupt Service Routines) are also made preemptible by implementing them as kernel threads. Consequently, the scheduling latency of real-time tasks is further reduced below tens of microseconds. This means that a real-time task can react to interrupts more quickly. The priority inheritance mechanism is also implemented. Thus, Ingo Molnar's modified Linux kernel meets all the functionalities of RTOS [12].

The performance of Complete Preemption was under investigation by previous studies [13, 14]. The evaluation results show that Complete Preemption was quite effective in real-time support because of its outstanding scheduling latency. However, our study reveals that Complete Preemption decreases the system throughput considerably due to many preemption points. The next section investigates Complete Preemption in more detail.

3 Linux Complete Preemption

In the preemptible Linux kernel, preemption is not possible inside critical sections due to synchronization. Consequently, a long scheduling delay for a real-time task is inevitable because the scheduling of the real-time task is postponed until the lock is released. Ingo Molnar's Complete Preemption replaced almost all the spin locks in the kernel with mutex-based spin locks. Using mutex-based spin locks, the real-time task can preempt any tasks even if those tasks are in a critical section. Other tasks trying to enter the critical section are enqueued in the waiter list of the lock. ISRs can be also preempted by real-time tasks in Complete Preemption since they are implemented using kernel threads. As a result, almost all the kernel codes are preemptible by real-time tasks. Complete Preemption also provides the priority inheritance mechanism which further enhances the responsiveness of real-time tasks.

One of the problems in Complete Preemption is that it sacrifices the system throughput in favor of shorter scheduling latency. Our preliminary study reveals that PREEMPT-RT, the Linux kernel patched using Complete Preemption, degrades the throughput considerably. Figure 1 shows the execution time of Hackbench [15] 50 and the number of context switchings during the benchmark tests on Pentium 2.4GHz machine. As shown in the figure, PREEMPT-RT takes about five times of the execution time compared to the Vanilla Linux kernel.



(a) Execution time (b) The number of context switchings

Fig. 1. Hackbench results of Vanilla and PREEMPT-RT

The significant decrease in the throughput is mainly due to excessive context switchings. In PREEMPT-RT, almost all kernel codes are preemptible by any interrupt even though the interrupt has nothing to do with real-time tasks. Unnecessary context switchings not only waste CPU cycles but also incur hidden costs such as TLB (Translation Lookaside Buffer) flush and cache misses.

4 Transparent and selective real-time interrupt services

PREEMPT-RT provides fast and stable scheduling latency. However, allowing a lot of preemption points adversely affects the overall performance since only real-time tasks need to preempt other normal tasks. In the following subsections, we describe our scheme to selectively service interrupts that are associated with real-time tasks.

4.1 Suppressing the preemptions by normal tasks

In Linux, all tasks are classified into two classes, a real-time class and a normal class. The Linux scheduler is based on the priority scheduling and tasks in the real-time class have always higher priorities than normal tasks.

PREEMPT-RT allows any task whose priority is higher than the currently running task to acquire the CPU even if the current task is in a critical section. This helps to reduce the scheduling latency for real-time tasks. However, the problem is that normal tasks are also able to preempt other normal tasks due to Complete Preemption. Since normal tasks are not so sensitive to the scheduling latency, context switchings from a normal task to another normal task take up the CPU cycles unnecessarily with additional overhead such as TLB flush and cache misses.

To remedy this problem, our scheme suppresses the preemptions caused by normal tasks as much as possible. Any higher-priority normal task does not preempt the current normal task immediately. Instead, the execution of the current normal task is guaranteed until the end of its time quantum in order

to avoid frequent context switchings between normal tasks. Note that real-time tasks still can preempt other tasks for the minimal scheduling latency.

In our implementation, when the Linux scheduler is invoked due to the change in the runnable task set, the scheduler first chooses the next task based on the Linux scheduling algorithm. If the next task is in the real-time class, the scheduler performs the preemption immediately. If not, however, the previous task is resumed to run until the end of the remaining time quantum, thus suppressing the preemption.

4.2 Selective handling of real-time interrupt threads

Because ISRs are implemented as kernel threads in PREEMPT-RT, they are scheduled by the Linux scheduler with their own priorities just like the other tasks. These interrupt threads are treated as real-time tasks. An interrupt thread can cause many context switchings, because it is able to preempt other tasks with the real-time priority.

In the proposed scheme, we basically treat interrupt threads as normal tasks. Although they have real-time priorities, we do not allow them to preempt other normal tasks in order to avoid the situation that the execution of a normal task is interrupted by non-critical interrupts.

The previous scheduling policy, however, may increase the scheduling latency considerably when some interrupt thread triggers a real-time task. Therefore, there should be a mechanism that we can somehow differentiate interrupt threads according to their relevance to real-time tasks. If we know a specific interrupt is associated with real-time tasks, we can set a special RT flag in the task structure of the corresponding interrupt thread. The RT flag indicates that the corresponding thread triggers a real-time task, hence it should be scheduled urgently.

Now the scheduling algorithm presented in Section 4.1 is modified as follows to deliver one or more predefined interrupts fast to real-time tasks. When the next candidate task is an interrupt thread, the Linux scheduler first checks whether the RT flag is marked in the task structure or not. If the RT flag is not set, the preemption is suppressed as explained in Section 4.1. On the other hand, if the RT flag is set, the interrupt thread is scheduled immediately to make the scheduling latency for real-time tasks short and stable.

4.3 Transparent association of interrupts with real-time tasks

The selective handling of real-time interrupt threads is effective to achieve the short and stable scheduling latency for real-time tasks while minimizing unnecessary context switchings between normal tasks. In the previous subsection, we assume that interrupts associated with real-time tasks are specified in advance by application developers. Often this assumption is reasonable in many real-time systems since real-time tasks are usually associated with specific sensors and actuators. However, it is annoying for application developers to specify the associated interrupts every time since they may not be familiar with hardware

details. Annotating the source code with a special system call not only lowers the application portability but also makes the application dependent on the hardware configuration on which it is running. Sometimes, it may not unclear which are right interrupts for the real-time tasks and the association may even change over time for complex real-time applications. In this subsection, we propose a way to transparently discover the relationship between real-time tasks and interrupts without any hints from application developers.

In PREEMPT-RT, a task is waken up in the function `try_to_wake_up()` when it is called by an interrupt thread. This means that if a real-time task is awakened by an interrupt thread, we can transparently identify those interrupts that are associated with real-time tasks without adding any kernel interface. The identified interrupt handler is marked with the RT flag in the task structure and the RT flag is later used by the Linux scheduler as described in Section 4.2.

In the Linux kernel, a part of interrupt handling can be delayed to *bottom halves*. The kernel thread called the *soft-irq* thread is usually used to perform the remaining work left by the interrupt handler. Since the PREEMPT-RT kernel also follows the same structure, the relationship between real-time tasks and interrupt threads may not be correctly recognized in `try_to_wake_up()` function if real-time tasks are waken up by bottom halves. We pay special attention to this case so that the original interrupt can be associated with the real-time task although the real-time task is awakened by the soft-irq thread.

The RT flag in the interrupt thread is removed when the associated real-time task either terminates or voluntarily turns into the normal task. Even though a real-time task terminates, some interrupt thread can be still marked with the RT flag since two real-time tasks may share the same interrupt. In this case, the RT flag is not removed until all the real-time tasks that share the interrupt terminate. Similarly, a single real-time task may be triggered by more than one interrupts, in which case all the interrupts are marked with the RT flag.

5 Evaluation

In real-time systems, the scheduling latency is one of the most important factors [13]. The overall throughput under limited computing resources is also an important characteristic of the system. Accordingly, the following two metrics are major concerns of our evaluation:

- **Latency**. The scheduling latency is the time between the event time and the start time of the task. In real-time systems, a stable scheduling latency must be guaranteed in various environments.
- **Throughput**. The throughput denotes the total amount of work that can be done in the given interval. In real-time systems, it is desirable to achieve the throughput as high as possible, while guaranteeing the predictable scheduling latency.

We describe our evaluation methodology in Section 5.1. Section 5.2 and Section 5.3 present our experimental results in detail.

5.1 Methodology

In this paper, we evaluate the following four Linux kernels:

- **Vanilla kernel.** The standard Linux kernel. Tasks are not preemptible while in the kernel.
- **Preemptible kernel.** The Linux kernel. Tasks are preemptible in the kernel area except critical sections.
- **PREEMPT-RT kernel.** The Linux kernel modified with Ingo Molnar's Complete Preemption.
- **Selective IRQ kernel.** The Linux kernel which implements the proposed scheme.

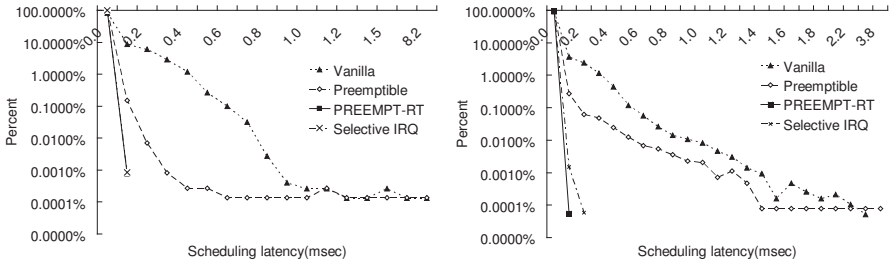
Note that our work was not compared to traditional real-time operating systems, such as QNX and VxWorks, because those RTOSes are limited in their functionality. The MontaVista Linux is largely similar to the Preemptible kernel.

We used an open source benchmark called *Realfeel* [16, 13, 17, 18] to measure the scheduling latency. We have also used two benchmark programs, *Hackbench* [15] and *Tbench* [19] to evaluate the throughput of the system. Hackbench and Tbench are executed as normal tasks. The experiments are performed on a 1GHz Nehemiah C3 Core VIA board with 256MB of RAM and a 40GB, 5400RPM IDE disk drive.

5.2 Latency

We generate a 256Hz stream of interrupts using Realfeel to measure the scheduling latency. The total 30 million interrupts are sampled by Realfeel, which run as a real-time task. To see how the scheduling latency is affected by other normal tasks, we give various stresses to the kernel. First, we executed two benchmark programs, Hackbench and Tbench, together with Realfeel (denoted as *light load*). To give heavier stress, one CPU bound task and one I/O bound task are added to two benchmark programs (denoted as *heavy load*). The CPU bound task is a matrix multiplication program and the I/O bound task is an FTP program that downloads ten 700MBytes files.

Figure 2 shows the scheduling latency under two types of system loads. The x-axis represents the scheduling latency in milliseconds and the y-axis the percent of the samples. In Figure 2(a), Selective IRQ shows the similar distribution of the scheduling latencies to that of PREEMPT-RT under the light load. Specifically, in both kernels, all samples are scheduled within 100 microseconds. Under the heavy load, the percent of the samples that have long scheduling latencies is increased in case of the Vanilla kernel and the Preemptible kernel as shown in Figure 2(b). Selective IRQ and PREEMPT-RT still have very stable latencies even under the heavy load. The maximum scheduling latency is measured to be less than 100 microseconds in both PREEMPT-RT and Selective IRQ.



(a) Under the light load (b) Under the heavy load

Fig. 2. The distribution of the scheduling latencies with 256Hz RTC interrupt

5.3 Throughput

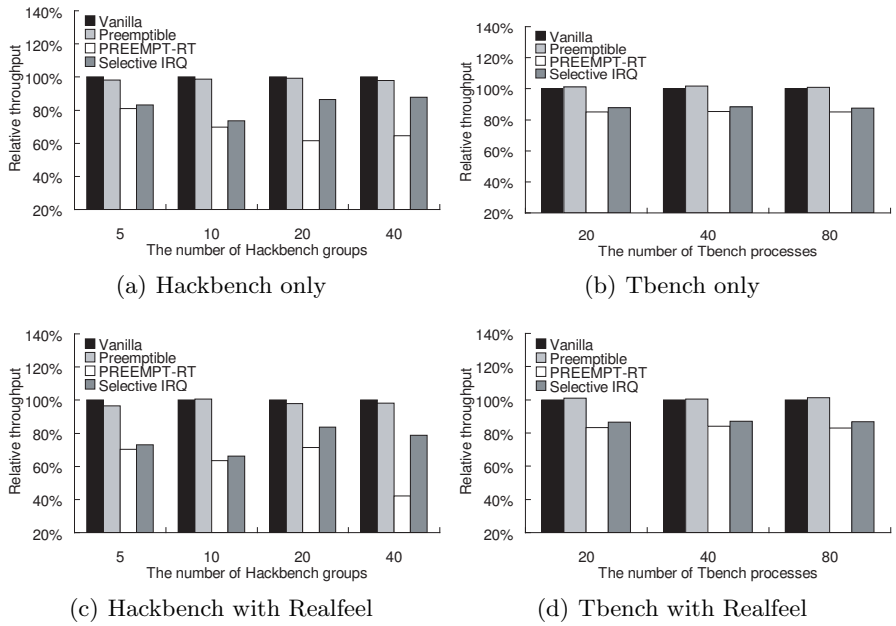
To observe the basic throughput without real-time tasks, we ran two benchmark programs mentioned in Section 5.1 while the system is idle. The next evaluation simulated a realistic workload, where the real-time task Realfeel is run simultaneously with other two benchmark programs. Recall that the scheduling latency of Realfeel in the latter case is already shown in Section 5.2.

Figure 3(a) and (b) present the benchmark results without other interfering tasks. All the values in the figure are normalized to the value of the Vanilla kernel. In Figure 3(a), the x-axis and the y-axis denote the number of Hackbench groups and the relative throughput of Hackbench respectively. PREEMPT-RT drops the throughput about 30% compared to the Vanilla kernel. The Preemptible kernel has the similar throughput to the Vanilla kernel but, as we have already seen in Section 5.2, it fails to provide the stable scheduling latency. Selective IRQ improves the Hackbench throughput by up to 40.3% compared to PREEMPT-RT as the number of Hackbench group increases.

Hackbench generates a considerable number of context switchings when many Hackbench groups run concurrently. The Selective IRQ improves the throughput because the proposed scheme intends to suppress unnecessary preemptions among normal tasks.

Figure 3(b) depicts the results of Tbench. The x-axis illustrates the number of Tbench tasks and the y-axis denotes the relative bandwidth of Tbench normalized to the result of the Vanilla kernel. This result also indicates that PREEMPT-RT achieves only 85% of the throughput of the Vanilla kernel. Selective IRQ enhances the throughput by up to 3.2% compared to PREEMPT-RT.

Figure 3(c) and (d) show the results of Hackbench and Tbench when a real-time task Realfeel is running together. From Figure 3(c), we can observe that the throughput of PREEMPT-RT becomes even worse with the presence of a real-time task. However, the proposed Selective IRQ scheme consistently shows the better throughput compared to PREEMPT-RT. Especially in Hackbench, PREEMPT-RT degrades throughput about 58% and Selective IRQ improves throughput up to by 86% compared to PREEMPT-RT. In Figure 3(d), the



(a) Hackbench only (b) Tbench only

(c) Hackbench with Realfeel (d) Tbench with Realfeel

Fig. 3. Hackbench and Tbench results

enhancement of the bandwidth under Selective IRQ is also increased from 3.2% to 4.6%.

6 Conclusion

In this paper, we proposed a novel scheme to improve the performance of embedded operating systems with real-time support. The performance degradation was mainly caused by a lot of preemption and scheduling points. Kernel-threaded interrupt service routines also contribute to the decrease in the performance. Excessive preemption and scheduling points yield many context switchings among tasks. The increased TLB flush and cache misses following the context switching are the direct sources of the performance degradation.

In the proposed scheme, the number of context switchings is reduced by the following two methods. First, we suppress the kernel preemptions caused by normal tasks as much as possible. Second, interrupts that are not relevant to real-time tasks are delayed. In order to provide prompt response to real-time tasks, our scheme transparently selects the interrupts that are related to real-time tasks and boosts them selectively. Consequently, the proposed scheme improves the system throughput significantly while exhibiting almost the same scheduling latency for real-time tasks.

Our work was implemented in the Linux 2.6 kernel. The experimental results show that the scheduling latency was below tens of microseconds. Moreover, the

throughput is enhanced by 40% compared to Complete Preemption when there is no real-time task. If there is a real-time task, the throughput is increased by up to 86%. We plane to optimize our scheme further to achieve better response time and throughput.

Acknowledgments. This research was funded by the MIC(Ministry of Information and Communication), Korea, under the ITRC(Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (IITA-2006-C1090-0603-0020). This work was also supported by DSRC(Defense Software Research center) and the Korea Science and Engineering Foundation(KOSEF) grant funded by the Korea government(MOST) (No. R01-2006-000-10724-0)

References

1. Yoffie, D., ed.: Completing in the Age of Digital Convergence. Harvard Business School Press (1997)
2. Abeni, L., Goel, A., Krasic, C., Snow, J., Walpole, J.: A measurement-based analysis of the real-time performance of linux. In: IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society (2002) 133
3. Molnar, I.: Complete preemption. http://people.redhat.com/mingo/realtime-preempt/ (2005)
4. Heursch, A.C., Grambow, D., Horstkotte, A.: Steps towards a fully preemptable linux kernel. In: Workshop on Real-time Programming. (2003)
5. Yodaiken, V.: The rtlinux manifesto. Proceeding of The 5th Linux Expo (1999)
6. Mantegazza, P., Dozio, E.L., Papacharalambous, S.: Rtai:real time application interface. Linux Journal (2000)
7. Dankwardt, K.: Real time and Linux, Part 3: Sub-kernels and benchmarks. Embedded Linux Journal **9** (2002) 33–37
8. Mercer, C.W., Tokuda, H.: Preemptibility in real-time operating systems. In: IEEE Real-Time Systems Symposium. (1992) 78–88
9. Wang, Y.C., Lin, K.J.: Enhancing the real-time capability of the linux kernel. In: IEEE Real Time Computing Systems and Applications. (1998)
10. Oikawa, S., Rajkumar, R.: Linux/rk: A portable resource kernel in linux. (1998)
11. Love, R.: The linux kernel preemption project. (http://kpreempt.sourceforge.net/)
12. Beneden, B.V.: Comp.realtime: Frequently asked questions (faqs) (version 3.6). http://www.faqs.org/faqs/realtime-computing/faq/ (2004)
13. von Hagen, W.: Real-time and performance improvements for the 2.6 linux kernel. Linux Journal (2005)
14. captain@captain.at: Linux real time patch review - vanilla vs. rt patch comparison. http://www.captain.at/howto-linux-real-time-patch.php (2006)
15. Russell, R.: Hackbench. (http://lkml.org/lkml/2001/12/11/19)
16. Hahn, M.: Realfeel. (brain.mcmaster.ca/ hahn/realfeel.c)
17. Webber, A.: Realfeel test of the preemptible kernel patch. Linux Journal (2002)
18. Williams, C.: Which is better – the preempt patch, or the low-latency patch? both! Linux Devices (2002)
19. Tridgell, A.: Dbench. (ftp://samba.org/pub/tridge/dbench/)