SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **39**:531–550
Published online 8 December 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.911

Catching two rabbits: adaptive real-time support for embedded Linux

Euiseong Seo[1,*,†], Jinkyu Jeong[2], Seonyeong Park[2], Jinsoo Kim[3] and Joonwoon Lee[3]

[1]*CSE Department, Penn State University, University Park, PA 16803, U.S.A.*
[2]*Korea Advanced Institute of Science and Technology, CS Department, Daejeon, Korea*
[3]*School of ICE, Sungkyunkwan University, Suwon, Korea*

SUMMARY

The trend of digital convergence makes multitasking common in many digital electronic products. Some applications in those systems have inherent real-time properties, while many others have few or no timeliness requirements. Therefore the embedded Linux kernels, which are widely used in those devices, provide real-time features in many forms. However, providing real-time scheduling usually induces throughput degradation in heavy multitasking due to the increased context switches. Usually the throughput degradation becomes a critical problem, since the performance of the embedded processors is generally limited for cost, design and energy efficiency reasons. This paper proposes schemes to lessen the throughput degradation, which is from real-time scheduling, by suppressing unnecessary context switches and applying real-time scheduling mechanisms only when it is necessary. Also the suggested schemes enable the complete priority inheritance protocol to prevent the well-known priority inversion problem. We evaluated the effectiveness of our approach with open-source benchmarks. By using the suggested schemes, the throughput is improved while the scheduling latency is kept same or better in comparison with the existing approaches. Copyright © 2008 John Wiley & Sons, Ltd.

Received 23 June 2008; Revised 21 October 2008; Accepted 22 October 2008

KEY WORDS: operating systems; embedded systems; real-time systems; soft real-time; Linux

*Correspondence to: Euiseong Seo, CSE Department, Penn State University, University Park, PA 16803, U.S.A.
†E-mail: euiseong@cse.psu.edu, euiseong@gmail.com

Contract/grant sponsor: Korea Science and Engineering Foundation (KOSEF)
Contract/grant sponsor: Korea government (MOST); contract/grant number: R01-2006-000-10724-0
Contract/grant sponsor: Defense Acquisition Program Administration
Contract/grant sponsor: Agency for Defense Development

1. INTRODUCTION

The functions of digital consumer electronics are being mixed together and the functional boundaries among those devices are becoming blurred. Naturally, the number of threads being executed simultaneously in the embedded systems for those devices are increasing, along with the computational complexity of each task.

Therefore, the traditional embedded operating systems that have light-weight architecture and a minimal set of functionalities are being replaced by fully-equipped embedded operating systems such as Windows CE or Embedded Linux, which are similar to their desktop operating system counterparts.

Embedded Linux is becoming popular in industry because its programming interface is compatible with that of its desktop counterpart and also has many of its state-of-the-art features. Its cost effectiveness resulting from its nature as an open-source product is another strong point. Such characteristics will reduce both the cost and the time in developing consumer electronic products.

However, the original Linux kernel was weak in supporting real-time threads because the context inside the kernel could not be preempted before Linux 2.6 was released. This caused delays in scheduling a thread even though it had higher priority than the currently running thread and that scheduling delay may have hindered scheduling a real-time application for which a timely response to the user command was critical for product quality.

The delayed scheduling of time-critical applications results in the malfunction of those applications. For example a music player that could not be scheduled on time makes the user hear an unpleasant noise or experience a sudden pause while listening. Considering that most applications related to multimedia, networking or GUI have mainly real-time requirements, the real-time support in embedded operating systems is an increasingly important issue. Therefore, many research projects have been conducted to add the real-time feature to the existing embedded operating systems.

The Linux kernel employed a basic preemptive kernel model with the release of Linux 2.6. Although the response latency was dramatically improved by the preemptive kernel model, the kernel could not yet be preempted within critical sections and the priority inversion problem also remained.

To resolve these issues Ingo Molnar released a kernel patch that is called the Complete Preemption patchset. In Complete Preemption, spin-locks, which are one of the synchronization mechanisms used in the kernel to guarantee mutual exclusion of critical sections, were replaced by mutexes. Spin-locks are unable to be preempted, while mutexes can be preempted. Therefore, even when a thread is working inside the kernel code, the thread could be switched to the other thread. Another significant difference is the design of thread handlers, which are changed to kernel threads from kernel functions. This improved design will ease context switches inside the interrupt handlers.

Therefore, most of the kernel context becomes preemptible and suppresses the scheduling latency within a certain bound. However, as a side effect this approach affects the throughput badly. With a simple experiment we verified that execution time for a benchmark could take time twice as long with the Complete Preemption Linux kernel as the original Linux kernel is configured not to support the kernel preemption.

The primary reason for the throughput degradation associated with the real-time features is the excessively increased context switches. The real-time Linux kernel will reschedule the threads or interrupt handler threads at every interrupt or scheduling point. Prioritizing the interrupt handlers required by some real-time threads is definitely necessary to handle the real-time threads on time because they can be executed after they are woken up by the interrupt handlers. However, context switching after the interrupts or events not related to the real-time threads just causes performance degradation, such as increasing TLB miss and cache miss ratio.

The performance degradation from supporting real-time scheduling is critical, especially in embedded systems in which performance is restricted usually by energy efficiency or manufacturing cost.

This paper suggests schemes to reduce throughput degradation while keeping scheduling latency within the degree of the Complete Preemption Linux kernel. We suggest a scheduling policy that suppresses context switches between two non real-time tasks. By doing so, we can achieve performance improvement with little increase of the scheduling delay for non real-time threads, which are tolerable to the scheduling delay. We also suggest a selective prioritizing of the interrupt handlers. Prioritizing only the interrupt handlers that affect the scheduling delays of real-time threads would decrease scheduling points and chances for context switches.

The rest of this paper is organized as follows. Section 2 introduces existing research activities on real-time support in Linux systems and analyzes throughput degradation of multitasking workloads under the Complete Preemption Linux kernel. Section 3 suggests our approaches to implement adaptive real-time scheduling and compares the pros and cons of each proposed approach. We implemented our schemes on the Linux kernel and evaluated them with open-source benchmark suites. The results are described in Section 4. Finally, we conclude our research in Section 5.

2. RELATED WORK

2.1. Real-time variations on Linux kernel

To support soft real-time thread scheduling, a general purpose operating system should have the following characteristics [1]: (i) a real-time operating system (RTOS) has to be multi-threaded and preemptible, (ii) the notion of thread priority must exist, as there is currently no deadline-driven OS, (iii) the OS has to support predictable thread synchronization mechanisms, (iv) a system of priority inheritance [2] has to exist, and (v) there should be no hidden OS behavior.

The hard real-time systems [3], generally having dedicated target thread sets, employ specialized thread schedulers that guarantee the predefined deadlines of those target tasks. Digital consumer electronics or computing systems for multimedia applications are open to third-party applications, which are unexpected at the design stage, and the data to be processed by those threads are also unexpected. Therefore, the deadlines for those threads cannot be determined easily in the design stage. That makes the hard real-time theory difficult to use in those systems.

As a result, as mentioned in the characteristic (ii), priority-based schedulers are preferred in those systems as alternatives to hard real-time schedulers. The priority scheduler should schedule the highest priority thread without delay right after the thread becomes ready to run. This will not yet guarantee deadlines or certain bounds of execution time for the tasks. However, this will provide the best response time the hardware can provide.

Priority-based schedulers are acceptable for most systems that have implicit deadlines. However, inventing a priority-based scheduler that perfectly obeys the sequence of priorities is not as straightforward as it sounds because there are shared resources in the operating system kernel.

Let us assume that a thread with the highest priority is activated and requires a shared variable to be released by a thread with the lowest priority, while a thread with the medium priority is scheduled to be executed. The desirable result would be that the scheduler executes the lowest-priority thread first and right after the lower-priority thread releases the locking mechanism of the shared resource, the thread with the highest priority should be scheduled. However, the schedulers that decide only with the apparent order in the priorities of the threads are not aware of this relationship among threads and therefore, schedule the medium-priority thread first, meaning that the thread with the highest priority would be delayed by the existence of the lower-priority task. This is a situation called *priority inversion* [2].

The condition (iv) states that a RTOS should have a solution for the priority inversion problem. For example, the well-known priority inheritance protocol should be prepared for all the shared resources such as functions, data structures as well as interrupt handlers.

The original Linux kernel has hardly been used as a RTOS because it does not have the attributes of (i), (iv) and (v). From the beginning, the Linux kernel was not designed for a real-time system. Vanilla kernels, the unmodified Linux kernels officially released from the Linux kernel developers community, did not have the kernel preemption ability nor the priority inheritance feature required to prevent priority inversion problem [4]. Thus, there has been much effort to add those attributes to the embedded Linux kernel.

There are two categories for existing real-time approaches: the sub-kernel approach and the preemptive kernel approach [5,6].

Figure 1(a) shows the sub-kernel architecture used in FSMLabs, RTLinux [7] and RTAI [8]. There are two kernels: a core kernel and a general purpose kernel, such as a Linux kernel in the sub-kernel architecture. A core kernel is a small and specialized kernel to execute real-time tasks. The real-time threads are executed only on the core kernel. Non real-time threads are handled by the Linux kernel that is scheduled to run by the core kernel when there is no real-time thread to execute.

The core kernel is designed to be preemptible and responsive. Therefore, it can suppress scheduling latency to be less than few μs. [6]. However, it generally provides minimized system calls and only essential hardware management functions. Therefore, the real-time applications

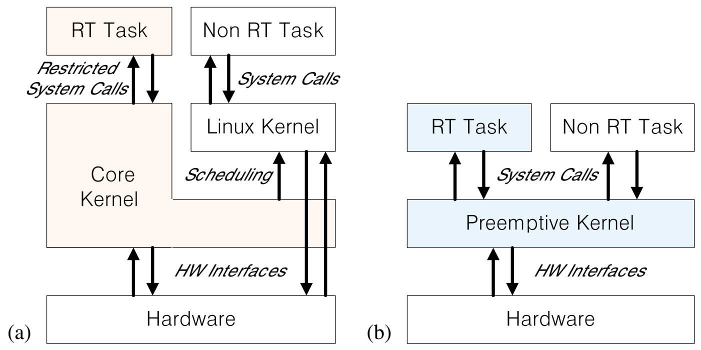


Figure 1. Kernel architectures for real-time systems: (a) sub kernel style and (b) preemptive kernel style.

should be implemented only with those limited operations and could not utilize the many useful libraries working in Linux systems.

In the preemptive kernel model [9], there is only the Linux kernel in the system that handles both real-time and non real-time threads as in Figure 1(b). The kernel used in the preemptive kernel model must be capable of preemption to the other user threads or the other kernel threads such as interrupt handlers, even in the kernel context. Unlike the sub-kernel approach, real-time threads can make full use of kernel features. The scheduling latency, however, can become unstable.

RED Linux [10] is an early example of a preemptive Linux kernel using the voluntary preemption approach. The kernel code of RED Linux has preemption points and the kernel context can be preempted only at those preemption points after checking the existence of pending interrupts. Therefore, even though the kernel could be preempted in many cases reducing the overall scheduling latency average, the length of the scheduling latency did not change. Jitters in the scheduling latency are less frequent, but could still occur in RED Linux.

From Linux 2.6 the Linux kernel provides the kernel preemption option [11]. The Linux kernel is preemptible when it is not in critical sections, enhancing the responsiveness of real-time tasks.

In addition to kernel preemption, Linux employs a scheduling policy based on the priority of tasks. There are two priorities for a task; the static priority, set by the user or the application explicitly, and the dynamic priority calculated at every scheduling point based on the tasks' static priority and the other parameters such as wake-up time and running time. When the scheduler function is called, the scheduler sets the next thread to the active thread with the highest dynamic priority.

Generally, the static priority is an integer value and less value means higher priority. The real-time threads have the static priority less than a threshold value. The threads with real-time priority will be scheduled before all the non real-time threads regardless of their dynamic priority values. Naturally, scheduling non real-time threads would occur only when there is no real-time thread in the run queue.

2.2. Complete Preemption kernel

Our approach to the real-time support in the Linux kernel is based on Ingo Molnar's Complete Preemption kernel patch [12,13]. Currently, it is one of the best real-time support variations on the Linux kernel.

A spin-lock, one of the synchronization mechanisms in the original Linux kernel, cannot be preempted. Therefore, as described in Section 2, context switching by a forced preemption is not possible inside critical sections in the preemptible Linux kernel due to the spin-locks. Consequently, a long scheduling delay for a real-time thread is inevitable because the scheduling of the real-time thread is postponed until the lock is released.

Complete Preemption replaced almost all the spin-locks in the kernel with mutex-based spin-locks, which could be preempted by other kernel threads. Using mutex-based spin-locks, the real-time thread can preempt any threads even if those threads are in one of the critical sections inside the kernel. Other threads trying to enter the critical section are queued in the waiting list of the corresponding lock.

Also, for the most part, interrupt handlers that were implemented as functions directly called by the processor through the interrupt vector table in the original Linux kernel could not be preempted because they disable the interrupts when they start, resulting in the priority inversion problem. However, the interrupt handlers can be also preempted by real-time threads 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 [2], which further enhances the responsiveness of real-time tasks.

Because interrupt service routines are implemented using kernel threads having static priorities, the priority inversion problem has been reincarnated. If real-time thread A, having highest priority in the system, awaits some event, the interrupt service routine for thread A must be scheduled. However, if there is some other thread B having middle priority between priority of thread A and priority of interrupt thread, scheduling of the interrupt thread is delayed after yielding CPU from thread B. Although the event for thread A has already arrived, thread A cannot promptly respond to the event because interrupt was not serviced. This is a case of interrupt inversion caused by assigning static priority to interrupt thread. Even with the careful assignment of the priorities for the interrupt handler threads, the priority inversion problems could not be eliminated perfectly.

2.3. Performance loss from real-time support

One of the drawbacks of Complete Preemption and other real-time supports using the priority-based scheduler is that they sacrifice the system throughput in favor of shortened scheduling latency. Our preliminary study reveals that the Complete Preemption patched Linux kernel degrades the throughput considerably. From now on we use PREEMPT-RT to denote the Linux kernel after being patched using the Complete Preemption patch set.

Figure 2 shows the execution time of Hackbench [14], which simulates the chatting service consisting of a server with multiple clients. There are multiple server/client groups that could be concurrently executed. Each server/client group in Hackbench has one server and 25 client threads and all the clients in a group send messages to the server of that group. Hackbench was configured with 50 server/client groups and the number of context switches during the benchmark run on the system with the Intel Pentium IV 2.4 GHz processor.

As shown in Figure 2, PREEMPT-RT takes about five times the execution time compared with the vanilla kernel denoted as Vanilla Kernel, which is the original Linux kernel without any patch. The significant decrease in the throughput is mainly due to excessive context switches. In PREEMPT-RT,

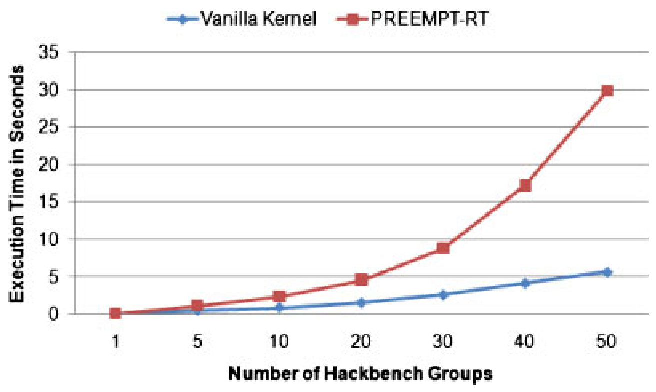


Figure 2. The execution time increases as the number of the client groups increases.

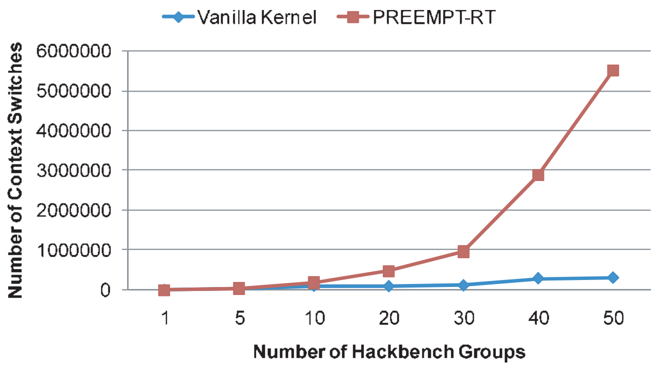


Figure 3. The number of context switching increases as the number of the client groups increases.

Table I. Costs from increased context switches in PREEMPT-RT with 40 groups of Hackbench.

	Exec. Time	Context switches	TLB misses	L2 cache misses
PREEMPT-RT : Vanilla	3.71	9.31	12.26	21.42

almost all kernel codes are preemptible by any interrupt even though the interrupt has nothing to do with real-time tasks.

As shown in Figure 3, the number of context switches linearly increased in Vanilla Kernel. However, it was dramatically increased in PREEMPT-RT. During sending and receiving of messages, the threads call many system calls and those unnecessary context switches not only waste CPU cycles, but also incur hidden costs, such as TLB flush and refill, along with increased L2 cache miss rate. Table I shows the changes in them by using PREEMPT-RT in comparison with the original kernel.

The context switches were done in PREEMPT-RT, usually when the scheduling function was called. The reasons for the increased context switches are as follows:

One reason is the scheduling policy that non real-time threads would be preempted by a thread with higher dynamic priority even when the currently running thread does not use up its time slice. The dynamic priority of a thread, which was just awakened, is boosted. Therefore, context switches happen at most scheduling points.

The other reason is that the scheduling points were increased heavily by prioritizing all the interrupts and events. Combined with the problematic scheduling policy, it even increased the number of context switches.

3. ADAPTIVE REAL-TIME SUPPORT

As stated in Section 2.3, although it has many improvements in scheduling latency, the Complete Preemption Linux kernel has seen significant throughput degradation, especially in the multitasking environment. Our aim is to enhance throughput under the Complete Preemption Linux kernel.

Our approach consists of the scheduling policy for suppressing the context switching between non real-time threads and three schemes for selective prioritizing of interrupt handlers. In addition, we also propose a priority inheritance algorithm for the interrupt handler threads based on the proposed schemes.

3.1. Suppressing preemptions by normal threads

The scheduler employs the priority-based algorithm and real-time threads that always have higher priorities than normal threads in it. The Complete Preemption patch allows any thread whose priority is higher than the currently running thread to acquire the CPU, even if the current thread is in a critical section. This helps to reduce the scheduling latency for real-time threads. However, the problem is that normal threads are also able to preempt other normal threads in the Complete Preemption Linux kernel.

Since normal threads are not so sensitive to the scheduling latency, context switches from a normal thread to another normal thread take up 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 threads as much as possible. Any higher-priority normal thread does not preempt the current normal thread immediately. Instead, the execution of the current normal thread is guaranteed until the end of its time quantum in order to avoid frequent context switches between normal tasks. Note that the real-time threads can still preempt other threads for the minimal scheduling latency.

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

3.2. Dynamic Scheduling Mode Change

The Complete Preemption Linux kernel uses the real-time supporting mechanisms regardless of whether or not a real-time thread exists. However, as shown in Section 2.3, it induces throughput degradation, and the degree of degradation increases as the degree of parallelism increases.

The performance loss from using the real-time supporting architecture is a worthless sacrifice when there is no real-time thread in use. Thus, it is reasonable to use real-time supporting mechanisms only when there is at least one real-time thread being executed in the system. Naturally, the traditional Linux scheduler will be used in the remaining time.

The kernel would recognize the existence of a real-time thread when there is a thread with a higher priority than a predefined threshold value. The priority of a thread is set with a system call *sched_setscheduler()* by the thread. Therefore, it is easy to identify a real-time thread.

The kernel has a counter variable to record the number of real-time threads in execution. The counter variable is increased when a thread is newly forked by the parent thread that is already a real-time thread, or the static priority of a thread is adjusted to a real-time value by calling *sched_setscheduler()*. Conversely, the counter variable is decreased when a real-time thread is terminated or the priority of a thread is adjusted to a non real-time value.

The dynamic mode change is illustrated in the flow chart illustrated in Figure 4. When the counter value is larger than 0, the kernel will follow priority-based scheduling. Therefore, the current running

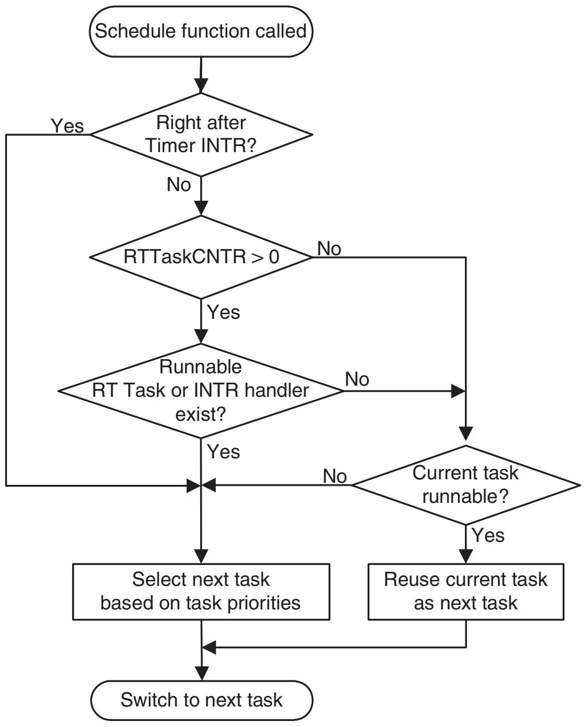


Figure 4. Flowchart of the scheduler following Dynamic Scheduling Mode Change.

thread will be switched to the real-time thread with the highest priority or the non real-time thread with the highest dynamic priority, when there is no real-time thread in the run queue.

Owing to the possibility that one of the non real-time threads has a role in the wake-up conditions of a real-time thread in sleep state, postponing the schedule of that non real-time thread could result in delayed awakening of the real-time thread. Therefore, not to affect the scheduling latency of real-time threads in sleep, the priority-based scheduler should be used whenever the counter value is larger than 0, even when there is no real-time thread in the running state.

When the counter value is 0, there should be no thread sensitive to the scheduling latency. Thus, in that case the throughput would be considered as a more important criterion than the latency. Therefore, we use the scheduling policy suggested in Section 3.1 to reduce the unnecessary context switches. Because there is no time-sensitive task in the system, even the interrupt handlers (except the timer interrupt) could be put off until the next tick when the timer interrupt arrives.

An instance of the timer interrupt could be a sign that the current thread used up its time slice and should yield to the processor of the other tasks. Thus, the timer interrupt should be handled on time.

3.3. Selective interrupt prioritizing

The scheduling of the real-time threads that run continuously could not be delayed, since the scheduler schedules only that thread continually as long as there is no real-time thread with the higher priority. The delay mostly occurs when the real-time thread is awaiting an interrupt in sleep state. Therefore, the delay in handling the interrupts that real-time threads await, results in the scheduling delay of that real-time thread. That is why all the interrupt handler threads should have the priority values of the real-time level.

Algorithm 1 Selective Interrupt Prioritizing

```
current returns currently scheduled task

intr_prioritize (interrupt_mode, interrup_group)
    current.interrupt_group ⟵ interrupt_group
    for ( ∀ interrupt handlers i ∈ interrupt_group)
        i.prioritized++

on thread exit ()
    for ( ∀ interrupt handlers i ∈ current.interrupt_group)
        i.prioritized- -

on schedule ()
    next ⟵ idle thread
    if (∃ runnable thread i where i.prioritized > 0)
        for (∀ runnable thread i where i.prioritized > 0)
            if (i.priority > next.priority)
                next ⟵ i
            endif
    else
        if (current still has unused time slices)
            next ⟵ current
        else
            next ⟵ the runnable thread with the highest priority
        endif
    endif
    context switch to next
```

However, in many cases, the possibility of interrupts affecting the scheduling of a real-time thread is small. For example, a thread waiting for the keyboard input will only be woken up by the keyboard interrupt. The interrupt from the network interface or timer interrupt has nothing to do with that task. Therefore, handling the interrupts from the timer and the NIC will not change the scheduling latency of that task.

Selective Interrupt Prioritizing described in Algorithm 1 is to prioritize the interrupt handlers related to the scheduling latency of real-time threads and to treat the other interrupt handlers as non real-time threads.

Before prioritizing the interrupts used by the real-time threads, the kernel should know which interrupts are used by real-time threads. This intuitive approach is that the real-time thread notifies the kernel of the interrupts it uses by calling a dedicated system call.

The interrupts are identified with their identification number, which is called *IRQ number*. However, except for some special devices such as a system timer or a keyboard, a device may have a different IRQ number depending on the system configurations. Also, in some cases multiple devices of different kinds share the same IRQ numbers. Therefore, to determine the interrupt for a device regardless of the system configuration, an identifier other than the IRQ number is required.

To solve this problem, *device group* was defined. Devices are grouped by their purposes, such as `NETWORK` for network related devices, `VIDEO` for the video cards and `HID` for human interface devices. The categories would be decided by the purpose of the target system. For example, a wireless network card could be categorized with Bluetooth network cards into `WIRELESSNETWORK` group, or it could be grouped with wired Ethernet cards into `ETHERNET` group. The choice will be dependent on the purpose and characteristics of the target real-time applications.

Each group has its interrupt modes, and there are two interrupt handling modes: one is prioritized and the other is normal. The interrupts that belonged to the prioritized group will be treated as if they have real-time priorities and the other interrupts will be handled as non real-time threads, except the timer interrupt hander.

A new system call *intr_prioritize()* will be used to notify the kernel of the required device groups. A real-time thread calls this system call at the initial stage and afterward, the scheduling of the interrupt handler threads in the required device groups will be treated with the real-time priorities originally assigned in the Complete Preemption Linux kernel. Therefore, the real-time thread will not have the scheduling latency longer than the Complete Preemption Linux kernel.

3.4. Transparent interrupt prioritizing

Selective prioritizing of interrupt threads is effective to achieve the short and stable scheduling latency for real-time threads while minimizing the unnecessary context switches between two non real-time threads.

However, it requires the real-time threads to notify their required device groups in advance. This means that the application developers should modify their source codes to add the system call *intr_prioritize()* in the initial stage of the applications and recompile the modified codes. As a result the final products are not compatible with the other Linux kernels. Therefore, we propose a new way to discover the relationship between real-time threads and interrupts required by those threads without any explicit hint from the developers.

Algorithm 2 shows the algorithm for finding the relationship between the interrupt handlers and the real-time threads and for prioritizing the interrupt handlers identified to be required by those real-time threads.

In the Linux kernel, a user-level thread is woken up by a function *try_to_wake_up()* during the execution of interrupt handlers. This means that if a real-time thread was awakened by an interrupt handler, that interrupt should be associated with the awakened real-time thread. The interrupt handlers identified to have relationships to the real-time threads should be prioritized to the real-time class.

In the Linux kernel, some parts of interrupt handler could be postponed to *bottom halves* [15]. In the Complete Preemption kernel, similar to the bottom halves, kernel threads called *soft-irq* threads are 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 threads and interrupt threads may not be correctly recognized in *try_to_wake_up* function if real-time threads are woken up by bottom halves. We pay special attention to this case so that the original interrupt can be associated with the real-time thread, although the real-time thread is awakened by the soft-irq thread.

Algorithm 2 Transparent Interrupt Prioritizing

```
current returns currently scheduled task

on wake_up (thread i)
    if (current is an interrupt handler)
        intr_prioritize (i, current)
        add_chain (i.alarm_chain, current)
    endif   current.prioritized++

on thread_exit (thread i)
    for (∀ i ∈ current.alarm_chain)
        i.prioritized- -

on schedule ()
    next ⟵ idle thread
    if (∃ runnable thread i where i.prioritized > 0)
        for (∀ runnable thread i where i.prioritized > 0)
            if (i.priority > next.priority)
                next ⟵ i
            endif
    else
        if (current still has unused time slices)
            next ⟵ current
        else
            next ⟵ the runnable thread with the highest priority
        endif
    endif
    context switch to next
```

The real-time priority of the interrupt thread is removed when the associated real-time thread either terminates or voluntarily turns into the normal task. Even though a real-time thread terminates, some interrupt threads can still maintain the real-time priorities, since two or more real-time threads may share an interrupt. In this case, the priority of that interrupt handler is not restored until all the real-time threads that share the interrupt terminate. Similarly, a single real-time thread may be triggered by more than one interrupt handler, in which case all the interrupts are marked to have the real-time priorities.

3.5. Priority inheritance of interrupt handler threads

As described in Section 3.1, we must carefully assign priorities to interrupt threads. In this section, we describe how to assign the priorities to interrupt threads to implement priority inheritance mechanism.

Algorithm 3 shows the priority inheritance mechanism for the interrupt handler threads. It adjusts the priority of an interrupt handler thread to the same value of the threads that wait for that interrupt.

Algorithm 3 Priority inheritance algorithm for interrupt handler threads

```
current returns currently scheduled task

on remove_wait_queue (wait_queue q, thread i)
    if (current is an interrupt handler & current ∈ q.irqth)
        remove_queue(q.chain, i)
        adjust_priority(q.irqth, max_priority(q.chain))
    else        q.irqth ⟵ current
    endif
on add_wait_queue (wait_queue q, thread i)
    add_queue(q.chain, i)
    adjust_priority(q.irqth, max_priority(q.chain))

adjust_priority(q.irqth, inherited_priority)
    for(∀ thread i ∈ q.irqth)
        if (i.base_priority > inherited_priority)
            i.effective_priority ⟵ inherited_priority
        else
            i.effective_priority ⟵ i.base_priority
        endif
    endif
```

The priority inversion related to the interrupt handlers occurs because the interrupt handlers get to compete with user-level threads for being scheduled and the priorities of the interrupt handler threads are fixed values regardless of the priorities of the user-level threads that wait for those interrupts.

The suggested algorithm uses the same method as Algorithm 2 for identifying the relationship between a user-level thread and interrupt handler threads which that user-level thread waits for. If a thread i was dropped from a wait queue q and *current* was an interrupt handler, then i should have waited for the interrupt of *current*. Therefore, if the priority of *current* was lower than that of i, it would be desirable to adjust the priority of *current* to the same value of i when i was an element in q.

A wait queue has a subsidiary queue *irqth* in addition to the main queue *chain*. *chain* stores the threads that sleep until certain kinds of events occur. *irqth* stores the interrupt handler threads that wake up the threads in the *chain* queue of that wait queue.

When a thread is inserted into a wait queue q, the algorithm calls *adjust_priority* for q. *adjust_priority* will set the priority of the interrupt handler threads in *q.irqth* to the highest value among the priorities of all the threads in *q.chain*. In the same manner when a thread is dropped from a wait queue q, the algorithm calls *adjust_priority* for q after updating *q.chain* to reflect the drop to the effective priorities of the interrupt handlers in *q.irqth*.

As a result by using Algorithm 3, an interrupt handler thread would have the highest priority among all the threads that wait for that interrupt and that would prevent the priority inversion.

However, the suggested algorithm is only effective when a real-time thread uses synchronous I/O. If a real-time thread used asynchronous I/O, it would not sleep and would not be inserted into a wait queue. Then the algorithm could not identify the relationship between the interrupt for that I/O and the real-time thread. Therefore, there would be no priority inheritance.

Undoubtedly in the case of using the asynchronous I/O, the caller thread does not sleep and therefore the scheduling latency of that thread has nothing to do with the scheduling latency of the I/O interrupt handler. However, considering that the I/O interrupt handler decides the response time of the I/O requests a thread makes, the I/O response time would remain in the same priority despite the high priority of the requesting thread. Therefore, the assignment of priorities to interrupt handlers should be carefully made to guarantee a satisfactory response time for I/O requests for the real-time threads by using the Selective Interrupt Prioritizing scheme.

There is another limitation of our approach regarding the priority inversion problem. When a resource is shared by a multiple number of threads that have different priorities, the interrupt handler for the resource could be prioritized by the highest priority of those threads even though the actual recipient of an instance of the interrupt is not the owner of the highest priority. However, considering that the execution time of an interrupt handler thread is usually much shorter than that of a user-level thread, the priority inversion between a user-level thread and an interrupt handler thread is a much less significant issue than that between two user-level threads, which occurs in cases without the proposed algorithm.

3.6. Practice of each proposed approach

Generally non real-time threads are not sensitive to the scheduling. Thus, modifying the scheduler following *suppressing the preemption by non RT tasks* does not harm the quality of applications. Although it prevents the instant preemption by non real-time tasks, the scheduling latency will remain in reasonable range, like non real-time Linux kernels, and there is no side effect to the scheduling latency of real-time tasks. Therefore, this policy can be used with other approaches most of the time.

Dynamic Scheduling Mode Change is a simple, transparent and easy to implement solution. However, it would not be effective when real-time threads and non real-time threads exist simultaneously since there would be no change between scheduling modes. Therefore, this would be the good choice for target systems with real-time threads that are rarely executed and have short lifetimes. However, this does not have any obvious overhead or side effect and could be used together with other techniques.

Selective Interrupt Prioritizing would be effective for cases in which the application set is fixed and real-time threads always coexist with non real-time tasks. It is also simple and without overhead. However, it needs the applications to notify their interrupt usage explicitly and, therefore, requires modification of the applications.

The system, with a fixed set of real-time threads that have long lifetimes, would be a good target for Selective Interrupt Prioritizing. Also, if a real-time thread has a short lifetime and is executed repeatedly, then Selective Interrupt Prioritizing would be the best among the three suggested algorithms.

Transparent Interrupt Prioritizing automatically detects the interrupts that affect the scheduling latency of real-time tasks. Therefore, it does not need any modification of existing applications.

However, in the beginning of a thread, some delayed scheduling incidents might occur because it needs time to detect all the relevant interrupts. Therefore, a thread that is frequently and repeatedly started and terminated might show significant scheduling jitters under Transparent Interrupt Selection for Prioritizing. To prevent those jitters, an application has to be designed not to be terminated but to sleep, while it is not being executed.

A system with many third-party real-time applications, whose presence is hard to be predicted at the design state, would be greatly helped by Transparent Interrupt Selection for Prioritizing because it does not require any modification of applications and still provides comparable reaction speed and throughput to Selective Interrupt Prioritizing.

Last, Dynamic Interrupt Priority Binding does prevent the priority inversion related to the interrupt handler threads. However, it has no effectiveness when there are not two or more competitive user-level threads, all of which have priorities higher than the interrupt handler threads, or there is no thread with a higher priority than the interrupt handler threads.

Although the condition for the priority inversion holds rarely, considering that Algorithm 3 is simple and, therefore, has negligible overhead, it would be desirable to employ it for systems that are open to their target applications.

4. EVALUATION

4.1. Evaluation environment

The purpose of the evaluation is measuring the scheduling latency and the variation of the throughput by applying the suggested schemes. The evaluation is done with the following kernels for comparison:

- *Vanilla*: The standard Linux kernel without any soft real-time support.
- *Preemptible*: The kernel compiled with voluntary kernel preemption feature, which was included in Linux 2.6.
- *PREEMPT-RT*: The Linux kernel compiled after applying Ingo Molnar's Complete Preemption patch.
- *Selective IRQ*: PREEMPT-RT kernel with Transparent Interrupt Selection for Prioritizing.

Among the suggested approaches, Dynamic Scheduling Mode Change does not have any change on the scheduling latency and also on the throughput, as long as there is at least a real-time thread in execution. Both the latency and the throughput would be the same as the PREEMPT-RT Linux kernel. In addition, Selective Interrupt Prioritizing differs from Transparent Interrupt Selection for Prioritizing only in the way to detect the interrupts to be prioritized. Therefore, we evaluated the scheduling latency and the throughput only for Transparent Interrupt Selection for Prioritizing, also with the suggested scheduling policy.

We used an open-source benchmark, *Realfeel* [16–18], to measure the scheduling latency. Realfeel issues periodic real-time clock (RTC) interrupts and measures the time needed for the computer to respond to these interrupts. Realfeel measures these response times and produces a histogram by putting the measurements into bins.

Along with the latency, the throughput that reveals the overhead to prioritize the interrupt handling should also be evaluated. For this, we used Hackbench and *Tbench* [19]. Tbench is a part of *Dbench*, which is an open-source benchmark suite that emulates the Disk I/O and the TCP

I/O of the well-known *Ziff-Davis NetBench* benchmark. Tbench consists of the workloads on the TCP I/O.

Evaluation was done on a system equipped with *VIA Nehemiah C3* 1.0 GHz processor, which is an energy-efficient embedded processor. It has 256 Mbytes of RAM, and all the workloads used in the evaluation did not require more than that amount of memory.

4.2. Latency

We generate a 256 Hz stream of RTC interrupts using Realfeel to measure the scheduling latency. The interrupts are sampled by Realfeel, which runs as a real-time task.

To see how the scheduling latency is affected by the other non real-time tasks, we give various stresses to the kernel. First, we executed two benchmark programs: Hackbench and Tbench, together with Realfeel (denoted as light load), respectively. To give heavier stress, one CPU bound thread and one I/O bound thread are added to the benchmark programs (denoted as heavy load). The CPU bound thread is a matrix multiplication program and the I/O bound thread is an FTP client that downloads 700 Mbytes files continually.

Figure 5 shows the scheduling latency under the light load. The *X*-axis represents the scheduling latency in milliseconds and the *Y*-axis the percentage of the samples. In Figure 5, Selective IRQ shows the same distribution of the scheduling latencies as that of PREEMPT-RT under the light load. Specifically, in both the kernels, all samples are scheduled within 100 μs.

Under the heavy load, the percentage of the samples that have long scheduling latencies is increased, in the cases of the Vanilla kernel and the preemptible kernel, as shown in Figure 6. Based on these results, we can tell that the real-time performance in the preemptible kernel feature included in Linux 2.6 is not adequate for time-critical applications.

Selective IRQ and PREEMPT-RT still have stable and short latencies even under the heavy load. The maximum scheduling latency is measured to be less than 100 μs in both PREEMPT-RT and Selective IRQ.

The extremely small difference between PREEMPT-RT and Selective IRQ is from the RTC interrupt before detecting the relationship between the Realfeel thread and the RTC interrupt handler thread.

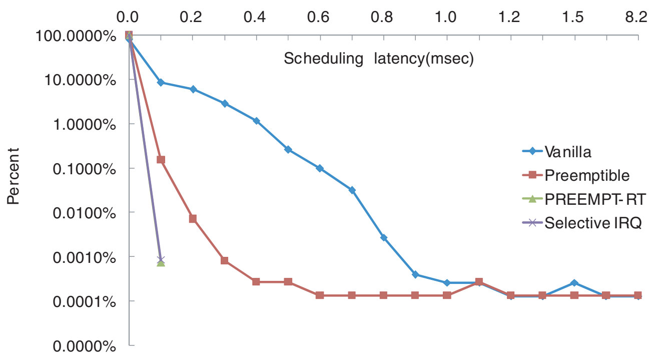


Figure 5. The execution time increases as the number of the client groups increase under light load.

4.3. Throughput

To evaluate the throughput of the kernels, the multiple incidents of each benchmark program are run simultaneously. For the evaluation with Hackbench, we change the number of server/client groups and we start the Tbench application multiple times for the evaluation with Tbench.

Figures 7 and 8, respectively show the throughput of Hackbench and Tbench normalized to those on Vanilla kernel,when Realfeel is running together with real-time priority.

Hackbench, configured to 40 groups, has 1000 threads that send messages to and receive messages from each other, concurrently. Sending and receiving generate the kernel traps to process the corresponding system calls, and at every kernel trap, there would be a schedule operation in PREEMPT-RT kernel. Sleeping threads waiting for the messages would wake up by receiving the messages and

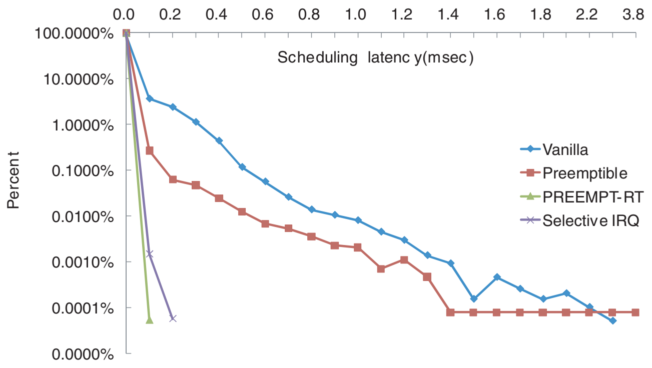


Figure 6. The execution time increases as the number of the client groups increase under heavy load.

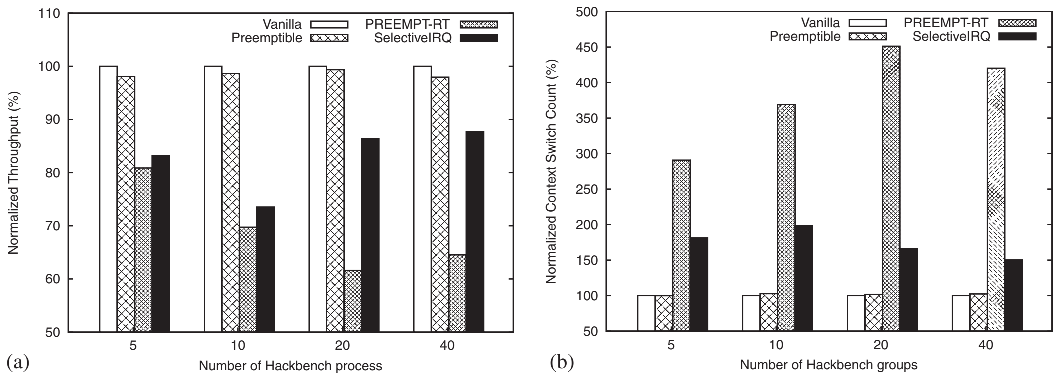
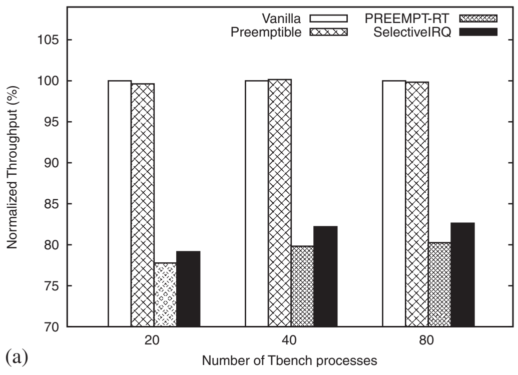


Figure 7. Throughput and context switch count of Hackbench with varying number of groups: (a) normalized throughput and (b) normalized context switch count.



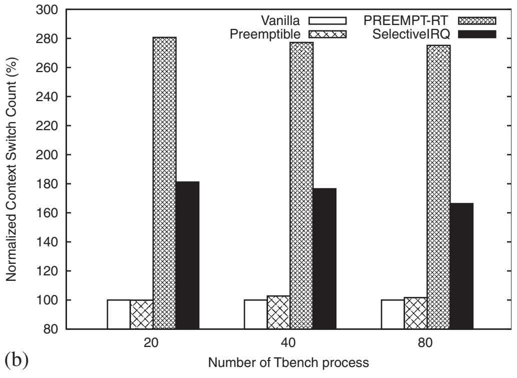


Figure 8. Throughput and context switch count of Tbench with varying number of concurrent instances: (a) normalized throughput and (b) normalized context switch count.

get boosted dynamic priorities. Therefore, the context switches will rapidly increase to the number of groups as shown in Figure 7(b).

Figure 7(a) shows that Selective IRQ gained significant performance improvement over PREEMPT-RT. This is due to the context switches between the non real-time user-level threads, which occur frequently by system calls, and synchronous I/O operations that are suppressed by the suggested scheduling policy. PREEMPT-RT degrades throughput to 61% and Selective IRQ improves throughput up to 25% compared with PREEMPT-RT. The bandwidth enhancement by Selective IRQ is from 3.2 to 4.6%.

Tbench has fewer threads concurrently running. Therefore, the effect to the L2 cache hit ratio would be less with the increased context switches. Even 80 concurrent running Tbench threads in PREEMPT-RT got 20% of the throughput decrease as shown in Figure 8(a). The suppressed context switches in Selective IRQ improved more than 2% of performance than PREEMPT-RT as shown in Figure 8(b).

4.4. Latency from priority inversions

To figure out the effectiveness of the priority inheritance mentioned in Section 3.5, we measured scheduling latency under intentionally generated priority inversion situation.

We run two tasks: one a real-time thread, which is an instance of Realfeel and the other is a thread doing matrix multiplications. The former thread has the highest priority in the system, and the latter thread has the priority between those of the real-time thread and the RTC interrupt handler thread that wakes up the real-time thread 128 times in a second. In other words, the priority of the RTC interrupt handler thread is the lowest among those three threads.

The matrix multiplication thread periodically wakes up every second and computes a high number of matrix multiplications, which takes about 400 ms. As described, the Realfeel thread awakes from the sleep state by 128 Hz RTC interrupts. The scheduling latency of the Realfeel thread is shown in Figure 9.

In Figure 9, Y-axis means scheduling latency of the Realfeel thread, and X-axis denotes flow of the system time. For the Realfeel thread about 0.4 ms of scheduling latency was observed every 1 s.

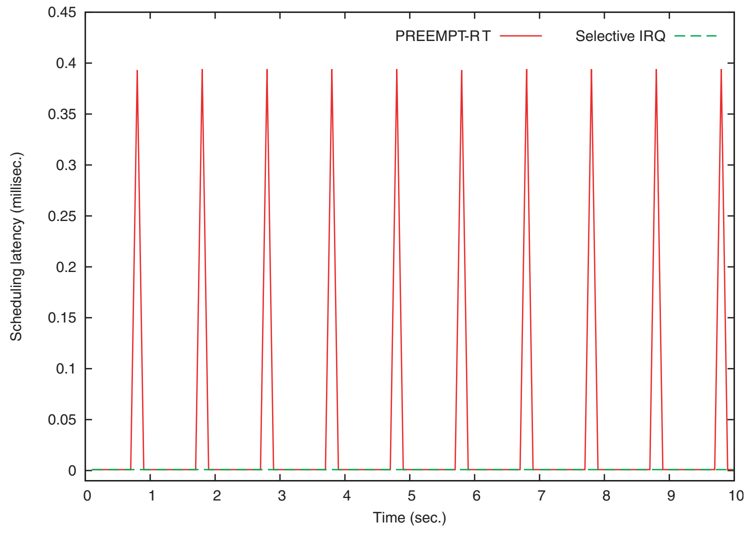


Figure 9. Scheduling latency under occurrence of priority inversion.

It is clear that this latency was due to the matrix multiplication thread and the RTC interrupt handler had to wait for the completion of the matrix multiplication computation.

Under Selective IRQ the RTC interrupt handler was bound to the Realfeel thread and therefore its priority was boosted to that of the Realfeel thread. As a result, the scheduling latency of the Realfeel thread always remains under.

5. CONCLUSION

Real-time support has become an essential feature for the embedded systems. However, it brings down the throughput, especially in multitasking systems, which are common in many digital devices. Its impact is also more critical in the embedded system than the other traditional computing systems because of their limited hardware performance.

We conducted evaluation for measuring the scheduling overhead induced from the real-time support and, based on our observations, we proposed practical solutions for resolving the performance degradation while keeping the real-time characteristics. As a bonus effect from the suggested schemes, the priority inversion problem related to the interrupt threads, which remains in the existing real-time Linux kernel, could be resolved by using an additional simple algorithm. Each suggested solution has its own pros and cons, and we analyzed both. The evaluation of the prototype implementation of the suggested schemes showed significant throughput enhancement.

The suggested schemes can be applied to any systems in which interrupt handlers are implemented as threads for preemptiveness. We believe that by using the suggested schemes, the embedded systems will get the real-time support with less sacrifice of performance than the existing solutions, and it will be helpful for the development of consumer electronics requiring massive multitasking.

ACKNOWLEDGEMENTS

This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MOST) (No. R01-2006-000-10724-0). This work was partially supported by Defense Acquisition Program Administration and Agency for Defense Development under the contract.

REFERENCES

1. Consortium AF. *Embedded Systems Design* (*Lecture Notes in Computer Science*, vol. 3436). Springer: Berlin, 2005; 258–286.
2. Sha L, Rajkumar R, Lehoczky JP. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers* 1990; **39**(9):1175–1185.
3. Liu CL, Layland JW. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM* 1973; **20**(1):46–61.
4. Man C-T, Li P, Li Y. Study of priority inversion in embedded linux. *The Proceedings of First International Conference on Innovative Computing*, *Information and Control*, Beijing, China, 2006; 217–219.
5. Dankwardt K. Real time and linux, part 2: The preemptible kernel. *Linux Journal* 2002; **8**.
6. Dankwardt K. Real time and linux, part 3: Sub-kernels and benchmarks. *Linux Journal* 2002; **9**.
7. Yodaiken V. The rtlinux manifesto. *The Proceedings of the Fifth Linux Expo*, Raleigh, NC, U.S.A., 1999.
8. Bianchi E, Dozio L. Some experiences in fast hard realtime control in user space with RTAI-LXRT. *The Proceedings of the Realtime Linux Workshop*, Orlando, FL, U.S.A., 2000.
9. Mercer CW, Tokuda H. Preemptibility in real-time operating systems. *The Proceedings of the IEEE Real-Time Systems Symposium*, Phoenix, AZ, U.S.A., 1992; 78–88.
10. Wang YC, Lin KJ. Enhancing the real-time capability of the linux kernel. *The Proceedings of the Fifth International Conference on Real-time Computing Systems and Applications*, Hiroshima, Japan, 1998; 11–20.
11. von Hagen W. Real-time and performance improvements for the 2.6 linux kernel. *Linux Journal* 2005; **2005**(134):8.
12. Molnar I. Real-time preemption patch. http://people.redhat.com/mingo/realtime-preempt/, 2005.
13. Heursch AC, Grambow D, Horstkotte A, Rzehak H. Steps towards a fully preemptable linux kernel. *The Proceedings of the Workshop on Real-Time Programming*, Lagowski Park, Poland, 2003.
14. Russell R. Hackbench. http://lkml.org/lkml/2001/12/11/19, 2001.
15. Rusling D. Bottom half handling. *The Linux Kernel*, Chapter 11.1. New Riders Pub, 2000; 139–140.
16. Hahn M. Realfeel. http://brain.mcmaster.ca/∼hahn/realfeel.c.
17. Webber A. Realfeel test of the preemptible kernel patch. *Linux Journal* October 2002.
18. Williams C. Linux scheduler latency. *Linux Devices*, white paper, March 2002. http://linuxdevices.com.
19. Tridgell A. Dbench. http://freshmeat.net/projects/dbench/, 2002.