Delayed Locking Technique for Improving Real-Time Performance of Embedded Linux by Prediction of Timer Interrupt

Jupyung Lee and Kyu-Ho Park
Computer Engineering Research Lab. EECS
Korea Advanced Institute of Science and Technology
Daejeon 305-701, Korea
jplee@core.kaist.ac.kr and kpark@ee.kaist.ac.kr

Abstract

In this paper, we propose a new technique, called a delayed locking technique, to improve the real-time performance of embedded Linux. The proposed technique employs the rule that entering a critical section is allowed only if the operation does not disturb the future execution of the real-time application. To execute this rule, we introduce the concepts of timer interrupt prediction and lock hold time acquisition. In addition, we designed and implemented a new high-resolution timer that is simple, yet efficient. We implemented the prototype on Linux 2.4.18. Experimental results show that the worst-case OS latency of real-time process is reduced to 23% of the original one, at the expense of slowdown of the non-real-time process by 20%. Though we focus only on embedded Linux, our technique is useful for all kinds of real-time operating systems in which the critical section is significantly long.

1. Introduction

In recent years, there has been a growing demand for handheld devices, such as PDA's and smart phones. Of the various available embedded operating systems that run upon such devices, the Linux operating system has attracted growing attention from handheld device vendors, due to its low cost, high reliability, support for various devices, and the availability of the source code.

One of the biggest problems to be solved with embedded Linux is to improve its real-time performance. Due to the explosive growth in demand for real-time applications, such as streaming audio/video, interactive games, and instant messaging, it has become essential for handheld devices to support such applications. To support such time-sensitive applications, the operating system must be able to preempt quickly any tasks currently executing when an interrupt occurs[6].

Unfortunately, standard Linux 2.4 does not allow preemption in kernel space. Depending on the tasks which run on background, the time it takes for a real-time task to start its execution after an interrupt occurs can reach up to 30 milliseconds, even when running on a powerful desktop PC[1]. Such a long latency significantly degrades the QoS of real-time applications, and this limitation may cause handheld device developers to rethink the adoption of Linux as the embedded OS for their devices.

However, with the advent of various software techniques and patches, it is now possible to make Linux fully preemptible, except when the spinlock is held, that is, when the kernel executes a critical section[7]. In Linux, the critical section is usually executed with the corresponding spinlock held. In general, the more complex a subsystem is, the longer the critical sections. Because Linux supports many complex subsystems, such as networking, file systems, and graphics subsystems, its critical section is usually longer than other light-weight real-time OSs[6]. This presents a serious obstacle toward a real-time Linux OS.

In this study, we try to solve this problem using the rule, "Entering a critical section is allowed only if the operation does not disturb the future execution of the real-time application." If one critical section is long enough to interfere with the real-time application, the corresponding locking is not allowed at that time, and the locking operation will be delayed until the execution does not disturb the real-time application. Therefore, we call this approach the *delayed locking technique*.

In order to execute this rule, we introduce *timer interrupt prediction* and *lock hold time acquisition*. We exploit the property that a timer interrupt is fully preemptible, in contrast to general interrupts. In addition, the lock hold time of each critical section is monitored constantly. In addition, we divide the timer interrupt into two classes in the hardware layer, urgent and normal timer interrupt, to distinguish one from the other.

The contribution of this paper is as follows:

Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05)
1080-1812/05 $ 20.00 IEEE

- The real-time performance of Linux is significantly improved by modifying only a small portion of the Linux kernel. We believe that the proposed technique will enable the adoption of the Linux OS for more embedded systems, which will reduce development costs and time-to-market. The proposed technique is also beneficial for all kinds of real-time operating systems in which the critical section is significantly long.
- Patches are available that can support a high-resolution timer, such as the UTIME patch[8]. Instead of reusing such a patch, we designed and implemented a new high-resolution timer patch that is simple, yet efficient.
- We implemented the prototype and analyzed the performance in an ARM-based embedded evaluation board. Most of the previous studies on real-time performance of Linux have been performed using a desktop PC. Our work focused on the embedded system from the beginning and our modified Linux kernel runs stable on the system.

2. Delayed Locking Technique

2.1. Basic Concept

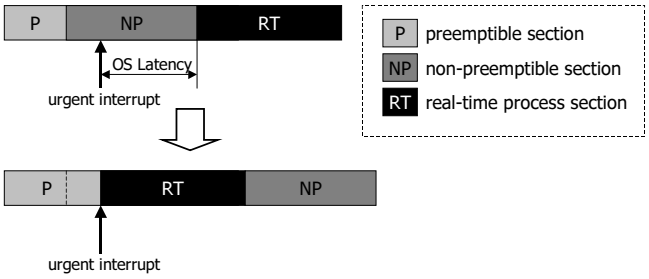

Figure 1. Basic concept of the delayed locking technique

The basic concept of the delayed locking technique is shown in Figure 1. An *urgent interrupt* is defined as an interrupt that will wake up the real-time process. If the real-time process that requires execution at time t is actually scheduled at time t', we can define the *OS latency* as L = t' - t. This definition comes from [1] and from now on, we focus on OS latency as the major performance metric.

If an urgent interrupt occurs during the execution of a non-preemptible section (that is, a critical section), the real-time process cannot be executed until the non-preemptible section has been completed. That causes OS latency, as depicted in above figure. If we can predict when an urgent interrupt will occur, and more specifically, whether it will occur earlier than the time when the kernel completes the execution of a non-preemptible section, then we can reduce the OS latency, by avoiding the locking and deferring execution of the section. In this case, the OS latency of real-time process is significantly reduced at the expense of delaying the execution of other processes.

2.2. Timer Interrupt Prediction

How can an interrupt be predicted? In general, it is impossible to predict the precise time when an interrupt will occur. In fact, an interrupt is usually used to handle jobs whose executions begin at unpredictable times, such as jobs that manage keyboard inputs, arrived packets, or data sent by hard disk drives.

However, in case of timer interrupts generated by programmable interval timer(PIT), we can fully predict the time of an interrupt occurrence. A timer interrupt occurs periodically (usually every 10 milliseconds in Linux), and so the occurrence time can be known with certainty. Even in case of a high-resolution timer, such as UTIME patch[8], which varies the period of the timer interrupt for the purpose of generating a more accurate timer interrupt for real-time processes, the occurrence time is still well known, because the next period is configured explicitly.

Many real-time applications depend on a timer interrupt for time-sensitive operation: Periodic real-time applications are reactivated by the timer interrupt. For example, Mplayer [10], the audio/video player, synchronizes audio and video frames and if two frames are not synchronized, then the system sleeps for a specific time and waits for a timer interrupt. Therefore, predicting only the timer interrupt is sufficiently beneficial to real-time applications. Dealing with other interrupt sources in a real-time operating system will be our future works.

One problem we have found is that all timer interrupts are not urgent interrupts: One timer interrupt occurs to reactivate the real-time process, while the other timer interrupt occurs periodically to update the system time of the OS. To distinguish one from the other, we divided timer interrupts into two classes, *urgent timer interrupt* and *normal timer interrupt*, in the hardware layer. When an urgent timer interrupt occurs, we can identify it as an interrupt attempting to wake up the real-time process; when a normal timer interrupt occurs, we can identify it as a 'periodic time tick'. A more detailed explanation of this is given in section 3.

2.3. Two Important Parameters : t_{ut_int} and $t_{lock,i}$

Two important parameters are defined here to further explain our delayed locking technique concept.

Firstly, the time remaining until the next urgent timer interrupt (t_{ut_int}) indicates the time remaining until the next urgent timer interrupt occurs. The parameter, t_{ut_int}, is easily calculated from TSC (Timer Stamp Counter) or a similar register. The TSC register, which is supported by Pentium CPUs, is a 64-bit counter that is incremented at each clock signal[9]. We can obtain t_{ut_int} by subtracting the current timestamp from the timestamp at which the next urgent timer interrupt will occur.

Secondly, the lock hold time ($t_{lock,i}$) indicates the time it takes to execute an i^{th} non-preemptible section. Each non-preemptible section has its own $t_{lock,i}$ value.

The time it takes to execute the same section may be different for the following reasons:

- Some sections have one entrance and several exits. Depending on the flow of execution, the lock hold time of these sections can vary.
- Even in the case of a section with one entrance and one exit, the amount of jobs performed inside the section may vary. For example, when a Linux kernel tries to send network packets, it first copies the data into the network buffer with the spin lock held. The longer the size of network packet, the longer the corresponding lock hold time becomes.
- If the interrupt occurs while running a non-preemptible section, the CPU automatically jumps to the starting address of the interrupt service routine(ISR) and performs the execution. Therefore, in that case, the ISR steals the execution time from the section and that increases the lock hold time of the section.

From among the measured values, we take either the maximum value or the exponential average value as the $t_{lock,i}$ parameter.

2.4. Locking Decision

Based on the two parameters mentioned above, we can determine whether locking will be allowed or prohibited. When one thread enters i^{th} non-preemptible section, it compares $t_{lock,i}$ with t_{ut_int}. If $t_{lock,i}$ is larger than t_{ut_int}, locking is prohibited and the thread enters a wait queue for future execution. Otherwise, locking is allowed and the thread starts the execution as usual. The decision process guarantees that the execution of a non-preemptible section will not delay scheduling the real-time process.

The decision process is illustrated in Figure 2.

2.5. Avoiding Starvation

Since our approach is to defer the execution of a non-real-time process conditionally, the starvation problem can arise if an urgent timer interrupt occurs frequently enough to cause the process to sleep indefinitely. More specifically, the condition under which the starvation problem may arise is as follows:

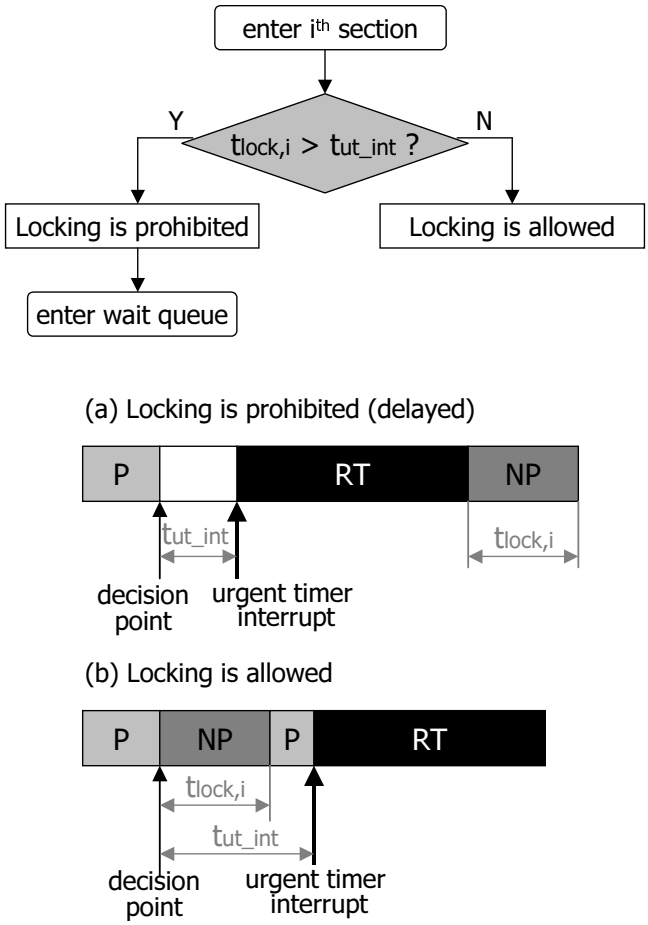

Figure 2. Locking decision process

$$p_{rt} < MAX(t_{lock,0}, t_{lock,1}, ..., t_{lock,N-1}) \quad (1)$$

, where p_{rt} is a period of a real-time process, $t_{lock,i}$ is a lock hold time of an i^{th} non-preemptible section, and N is the number of non-preemptible sections.

The condition satisfying the above equation does not necessitate that the non-real-time process will fall to starvation. However, once the process tries to enter a non-preemptible section whose lock hold time is longer than p_{rt}, the process will sleep indefinitely, until the real-time process has finished or the period has been increased.

One possible solution to avoid such a starvation is to limit the minimum allowed period of a real-time process to be as follows:

$$p_{rt,min} = MAX(t_{lock,0}, t_{lock,1}, ..., t_{lock,N-1}). \quad (2)$$

For example, if $p_{rt,min}$ = 2 milliseconds and one real-time process wants to be scheduled every 1 millisecond, it is not fully allowed; instead, the process will be scheduled every 2 milliseconds.

Another solution is that if the non-real-time process is blocked by an urgent timer interrupt more than, for example, 100 times, then the process ignores the locking decision and forces itself to acquire the spin lock. At that very moment, the real-time process may experience a longer latency than before, due to the long non-preemptible section. However, this situation occurs rarely, because even in the worst case, such a long latency will not occur for the next 99 times. In addition, starvation is avoided, since non-real-time process will have already passed the obstacle, a non-preemptible section.

All the aforementioned problems arise from the significantly long non-preemptible section of the Linux OS. Many Linux kernel developers have made efforts to break the spin lock section into smaller sections[1][4][5][11]. Since such approaches are independent of ours, as the spin lock sections become smaller, we can take full advantage of the development, and the starvation problem of our technique will become less severe.

3. High Resolution Timer

A standard Linux timer is triggered by a periodic timer interrupt, which is generated by a PIT, and the period is usually 10 milliseconds. A problem arises because the resolution of the timer, 10 milliseconds, is not sufficient for the time-sensitive real-time applications. For example, consider a periodic task that needs to execute every 100 μs. The task expects that the timer interrupt will wake-up itself every 100 μs; however, it cannot avoid the latency caused by the low resolution of the timer, which may reach up to 10 milliseconds in the worst-case scenario.

The easiest solution for this problem is to reduce the period of the timer interrupt. However, frequent timer interrupts will increase the overhead to handle the interrupt. An alternative is to adjust the period only when there is an active task that needs to be scheduled accurately. This approach has been adopted for UTIME patch[8].

Instead of reusing this patch, we designed and implemented a new high-resolution timer, which is simple and efficient. The new implementation exploits two timers: one timer generates a normal timer interrupt, while the other generates an urgent timer interrupt for real-time processes. In normal cases, only the normal timer is activated, and it generates a periodic normal timer interrupt. The urgent timer is activated only when a real-time process demands a high-resolution timer; it generates an urgent timer interrupt at a specific time. Since most of the commercial CPUs support multiple timers, such an implementation is possible.

In our study, Samsung S3C2410 microprocessor (ARM920T core)[14] is used for the embedded system, and it supports four timer modules. Among the four timers, one is used as an urgent timer and another as a normal timer. Figure 3 and figure 4 illustrate two kinds of high-resolution timers: UTIME and our implementation.

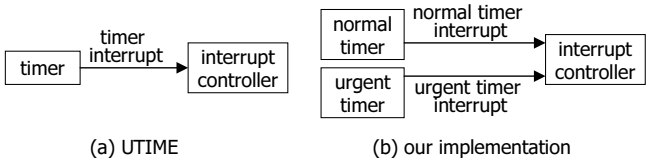

Figure 3. High-resolution timer

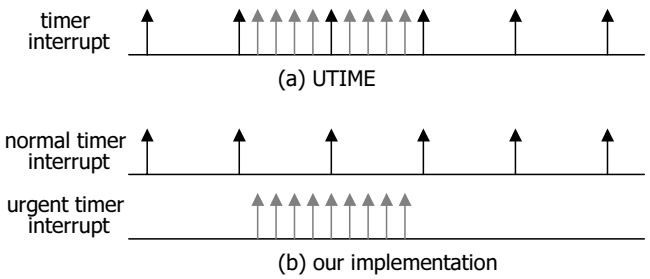

Figure 4. High-resolution timer : example

The advantages of our implementation over UTIME are as follows:

- The implementation is simplified. We do not modify the flow of execution, that starts from a normal timer interrupt, and includes updating the system time(jiffies), reorganizing the timer list, finding an expired timer, reducing the CPU quantum of the current process, and if necessary, rescheduling. We simply add another flow of execution that starts from an urgent timer interrupt. When an urgent timer interrupt occurs, the only thing that the handler of the interrupt has to do is to wake up the real-time process that needs to be scheduled at that moment. By dividing the timer interrupts into two classes, the handler of a normal timer interrupt does not have to be modified and only the handler of an urgent timer is newly implemented.
- Thanks to the small amount of jobs performed inside the urgent timer interrupt handler, the handler can reac-

tivate the real-time process very quickly. More specifically, in UTIME, the bottom half handler of a timer interrupt reactivates the real-time process, while in our implementation, the same job can be performed inside the interrupt service routine (ISR) of an urgent timer interrupt. In our implementation, an urgent timer interrupt does not need the bottom half handler, because the amount of the job is small enough to fit into the ISR.

- Since urgent and normal timer interrupt signals are separated, the interrupt controller is instantly aware of whether an arrived interrupt signal needs to be serviced immediately or not. Therefore, interrupt prioritization is possible: the interrupt controller grants highest priority to an urgent timer interrupt. By doing so, the responsiveness of the real-time process is improved.

If a real-time process wants to use high-resolution timer, it calls the newly added system call (*microsleep*). This system call turns on the urgent timer, which will generate an urgent timer interrupt after a specified time.

4. Overall Architecture

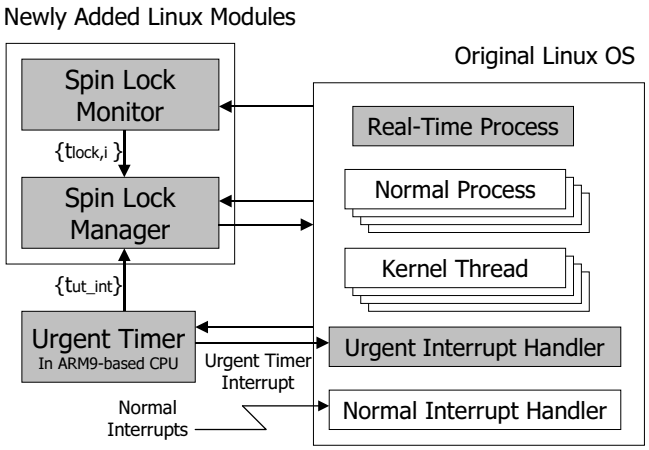

Figure 5. Overall architecture

The overall architecture of our prototype is shown in Figure 5. The gray colored rectangles represent the portions that are related to delayed locking. Two Linux modules, a *spin lock monitor* and a *spin lock manager*, are added to original Linux OS, and an *urgent timer*, which is implemented inside the S3C2410 CPU(ARM9TDMI core), is newly activated. The spin lock monitor continuously tracks the lock hold time of each non-preemptible section, and if necessary, it updates the $t_{lock,i}$ value. The spin lock manager determines whether it will allow the locking or not, by comparing t_{ut_int} from an urgent timer and $t_{lock,i}$ from the spin lock monitor, as explained in section 2.4.

Whenever one process tries to acquire a spin lock, it first has to get permission from the spin lock manager. Once the permission has been issued by the spin lock manager, the process acquires the spin lock and starts the execution of the non-preemptible section.

And urgent timer generates a urgent timer interrupt at a specific time if a real-time process demands it, as mentioned in section 3. Once the urgent interrupt occurs, the *urgent interrupt handler* reactivates the corresponding real-time process. Other interrupts, including a normal timer interrupt, are serviced as usual by the normal interrupt handler.

Currently, in a standard Linux, acquiring a spin lock is allowed anywhere inside the Linux kernel source code, including all kinds of device drivers and kernel threads. This means that a poorly designed device driver making one lock hold time significantly long, can degrade the real-time performance of the overall system. Our architecture, which introduces a centralized spin lock manager, resolves such problems efficiently.

5. Implementation

```
#define spin_lock(lock)
do {
        preempt_disable();
        _raw_spin_lock(lock);
}
```

```
#define spin_unlock(lock)
do {
        _raw_spin_unlock(lock);
        preempt_enable();
}
```

Figure 6. spin_lock() and spin_unlock() in preemptible Linux

```
#define spin_lock(lock, i)
do {
      if (t_lock[i] > t_intr) {
        sleep_this_thread();
      }
      measure_current_time();
      preempt_disable();
      _raw_spin_lock(lock);
}
```

```
#define spin_unlock(lock, i)
do {
    update_lock_hold_time(i);
    _raw_spin_unlock(lock);
    preempt_enable();
}
```

Figure 7. spin_lock() and spin_unlock() in our implementation : pseudo-code

We implemented the prototype by modifying Linux 2.4.18 on ARM-based embedded evaluation board. We first added a preemptible Linux patch[1][4] to a standard Linux 2.4.18 kernel. As mentioned earlier, preemptible

Linux makes itself preemptible except when the spin lock is held. To implement this, the patch extends the lock acquisition/release function, as illustrated in Figure 6. The functions *_raw_spin_lock()* and *_raw_spin_unlock()* point to original spin_lock() and spin_unlock() functions, respectively.

The spin lock was originally introduced in Linux to support multiprocessor systems, in other words, to protect shared resources from being concurrently accessed by kernel control paths that run on different CPUs [9]. Thus, in uniprocessor systems, such as our system, *_raw_spin_lock()* and *_raw_spin_unlock()* do nothing, and therefore, *spin_lock()* and *spin_unlock()* act only as the preemption markers. We would like to mention that spin locks in Linux running on uniprocessor systems are not 'real' spin locks: locks that a processor continuously tries to acquire spinning around a loop.

We extend these functions further to support delayed locking. The pseudo codes are shown in Figure 7. By adopting such an approach, we have modified a very small portion of the Linux source codes.

One problem that arose during the implementation is that scheduling (calling *schedule()* function) is not allowed in Linux during the execution of either the soft IRQ or the bottom half handler. Therefore, even if the soft IRQ or bottom half handler tries to hold the spin lock and it is prohibited by spin lock manager, it is impossible for the current thread to enter the wait queue. This would cause overall Linux kernel to halt.

To resolve this problem, we changed the soft IRQ and bottom half handler into kernel threads. By doing so, scheduling can occur during the execution of either the soft IRQ or the bottom half handler. Similar techniques are employed in [12] and [13].

In addition, we have modified Linux further to support a high-resolution timer. The modification includes an urgent interrupt service routine (ISR) and a new system call interface (as mentioned in section 4).

The modified Linux runs on an ARM-based embedded evaluation board. The specification of the board appears in Table 1.

The CPU, a Samsung S3C2410 (ARM920T core), supports 4 timer modules, and we use 2 timers: one for the urgent timer and one for the normal OS timer. The Pentium CPU supports TSC (Time Stamp Counter), while S3C2410 supports TCNTO (Timer Count Observation Register)[14], which stores the remaining number of clocks until the next timer interrupt. Therefore, by reading this register, we can calculate t_{ut_int}.

CPU	Samsung S3C2410
Core	ARM920T
I-Cache	16kB
D-Cache	16kB
Oper. Freq.	200MHz
DRAM	64MB
Network	10Mbps Ethernet

Table 1. Specification of ARM-based embedded evaluation board

6. Experiments

6.1. Experiment Setup

We made two processes run simultaneously on our modified Linux kernel: one was a real-time process and the other was a background process. The real-time process models a periodic real-time application, such as video player, and it iterates the following loop for 10,000 times:

1. Reads the current time t_1
2. Sleeps for a time T
3. Reads the current time t_2

In an ideal case, t_2 is equal to t_1 + T; in other words, OS latency is 0.

The background process tries to disturb the reactivation of the real-time process, by causing frequent interrupts and thus triggering long non-preemptible sections. More specifically, the background process repeatedly transfers a series of packets through TCP/IP, which makes the non-preemptible section significantly long.

We measured the OS latency, which was defined in section 2, during each iteration of the real-time process. Of course, as ISR latency and OS latency are close to 0, the real-time performance of the system is said to be better.

6.2. Major Non-preemptible Sections Caused by a Background Process(TCP/IP)

First, we tried to find a non-preemptible section caused by the background process, that is long enough to degrade the real-time performance significantly.

By analyzing the Linux kernel, particularly the TCP/IP network stack, we found three major non-preemptible sections in our modified Linux kernel, and these are shown in Table 2.

Next, we repeatedly measured the lock hold time of each section for 10,000 times.

Figure 8, 9 and 10 show the measured lock hold time of each lock. The interesting observation from these results is

$lock_i$	location	description
$lock_0$	do_softirq()	handles soft IRQs
$lock_1$	tcp_sendmsg()	sends packets via TCP/IP
$lock_2$	net_rx()	handles received packets

Table 2. Major non-preemptible sections caused by the background process(TCP/IP)

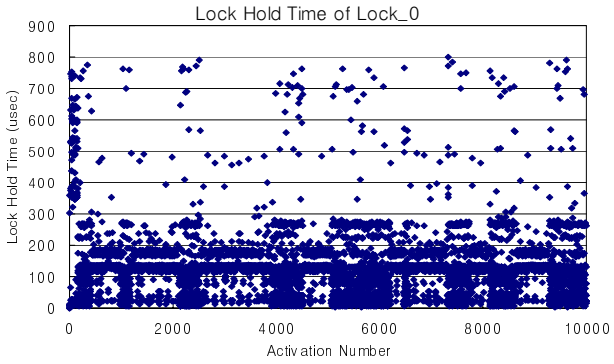

Figure 8. Lock hold time of $lock_0$

the clustering of lock hold times into a few groups. In Figure 9, for example, many points are clustered in approximately 1900, 1300, 300 μs. This indicates (i) that the time taken to execute the same section may be different, according to several factors, as mentioned in section 2, and (ii) that each group of points is caused by a different flow of execution.

From among the measured values, we took the maximum value to be $t_{lock,i}$, as mentioned in section 2. The $t_{lock,i}$

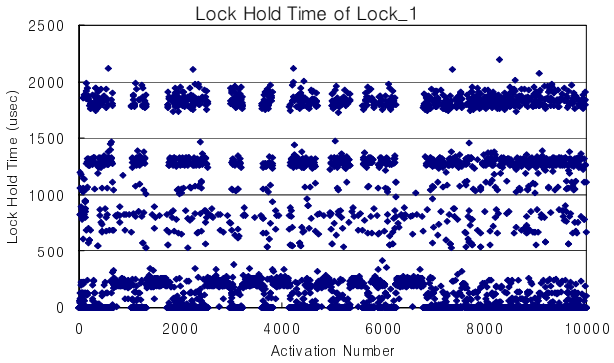

Figure 9. Lock hold time of $lock_1$

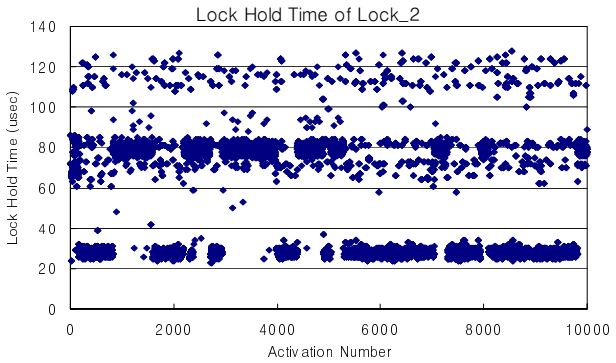

Figure 10. Lock hold time of $lock_2$

value of each non-preemptible section appears in Table 3.

$lock_i$	average time	maximum time	$t_{lock,i}$
$lock_0$	121 μs	800 μs	800 μs
$lock_1$	352 μs	2192 μs	2192 μs
$lock_2$	56 μs	128 μs	128 μs

Table 3. The $t_{lock,i}$ value of each non-preemptible section in our experiment

A problem arises when we take maximum value: even when the actual lock hold time is not long enough to interfere with real-time process, the locking may be prohibited by the spin lock manager. However, by taking the maximum value, we can guarantee that the real-time process will not collide with the corresponding non-preemptible section, and this is why we chose this option. If we can predict the actual lock hold time before entering the section, the spin lock manager can always make the correct decision, which will be our future works.

6.3. OS Latency

Based on the parameters in Table 3, we conducted experiments both with delayed locking and without delayed locking, and measured the respective OS latencies. To avoid starvation, we configured the period of the real-time process to be 4000 μs. This value is sufficiently larger than maximal value of $t_{lock,i}$, which is 2192 μs, and therefore, it does not meet the starvation condition shown in equation (1). Figure 11 and Figure 12 show the result of each case, respectively. The OS latency is significantly reduced when using the delayed locking technique.

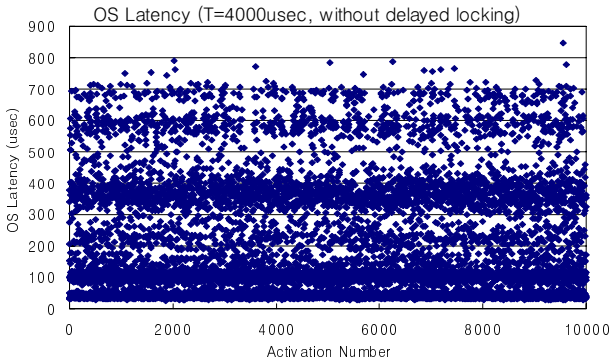

Figure 11. OS latency : without delayed locking technique

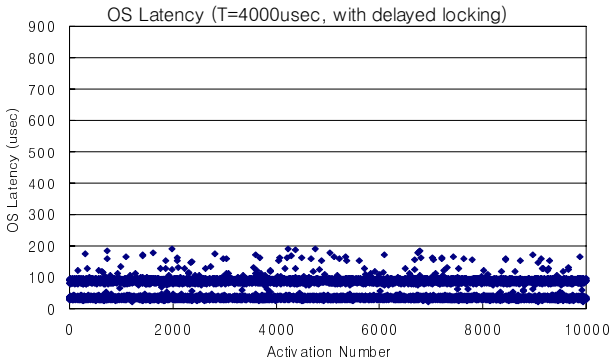

Figure 12. OS latency : with delayed locking technique

The minimum, maximum, and average values of OS latency for the two cases are compared in Table 4. The worst-case response time is the most important performance metric for a real-time system. According to this table, the worst-case OS latency is reduced to 23% of the original one. The average case is also reduced to 32% of the original one.

6.4. Effects of Delayed Locking Technique on Non-real-time Process

As mentioned in section 2, such performance improvement is achieved at the expense of delaying the execution of other process, in this case the background process, which keeps on sending a series of packets. In order to investigate the effect of the delayed locking technique on non-real-time process, we measured the performance of the background process in terms of 'transfer time per 1Kbytes', in other words, the time taken for the process to send 1Kbytes of data.

	w/o delayed locking	w/ delayed locking
Minimum	24 μs	22 μs
Maximum	847 μs	192 μs
Average	197.14 μs	63 μs

Table 4. OS latency comparison : with and without delayed locking technique

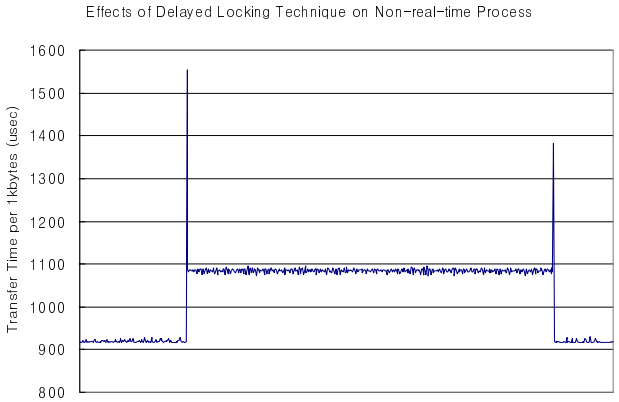

Figure 13. Effects of delayed locking technique on non-real-time process

The result of the measurement is shown in Figure 13. High transfer time(low throughput) in the middle of the execution is caused by running the highest-priority real-time process. This indicates that even though the performance degradation of background process by 20% is imposed by the real-time process, the process still runs well with a stable throughput, and no starvation arises. We would like to stress that the slowdown of the non-real-time process caused by the delayed locking technique can be reduced by using a lock hold time prediction, as mentioned above, or by breaking the lock section into smaller sections.

7. Related Work

Many studies have been conducted to improve the real-time performance of the Linux OS or other general-purpose operating systems. Among them, in particular, there are some implementations for reducing the OS latency of Linux, and these can be classified into two approaches.

The first approach, called a *patch approach* herein, makes standard Linux OS preemptible. *Low-Latency Linux*[16] inserts explicit preemption points inside the non-preemptible kernel. When one thread passes the points, it can explicitly decide to yield the CPU to other threads. *Preemptible Linux*[7] makes the kernel fully preemptible except when the spinlock is held. This patch has been added to the 2.5.4 kernel. *Preemptible Lock-Breaking Linux*[1] breaks long spinlock sections into smaller sections. This facilitates the reduction of OS latency, but finding a 'breakable' section and breaking it while maintaining OS consistency is a cumbersome and time-consuming job. We implemented the prototype based on preemptible Linux[7], which was integrated by the delayed locking technique and the high-resolution timer.

The second approach, called *dual-kernel approach*, runs two different kernels simultaneously in different layers: in the lower layer runs a real-time kernel, and over the layer runs Linux, including not only the Linux kernel but also Linux processes, as a background process. In this case, real-time tasks do not run on the Linux kernel, but on the real-time kernel, which makes it possible for the real-time tasks to preempt any tasks quickly no matter how significant a load the Linux kernel imposes on the system. RTLinux[2] is a widely used real-time operating system based on the dual-kernel approach.

In contrast to either standard Linux or preemptible Linux, RTLinux allows kernel preemption even when Linux is in the middle of a spin lock section. This is because RTLinux does not allow real-time processes to dynamically request memory, share spin locks, or synchronize on any data structures [15]. In other words, RTLinux does not allow mutual resource sharing between the real-time processes and the Linux OS. Because of such a constraint imposed on real-time processes, a real-time process never has to wait for the Linux side to release any resources; however, a great portion of the real-time application source codes should be re-written to meet this constraint.

Our approach, preemptible Linux integrated with the delayed locking technique, is different from dual-kernel approach in that no special constraint is imposed to implement a real-time process. Therefore, configuring an application, such as a web server or a streaming player, as a real-time process is relatively easy. Therefore, our approach is more suitable for improving the responsiveness of some applications in a general purpose operating system, while a dual-kernel approach is more suitable for constructing a hard real-time system.

8. Conclusion

In this paper, we have proposed a delayed locking technique for improving the real-time performance of embedded Linux. This technique employed the rule that entering a critical section is allowed only when the operation does not disturb the future execution of the real-time application. To implement this rule, we have introduced interrupt prediction and lock hold time acquisition. In addition, we designed and implemented a new high-resolution timer, which is simple and efficient.

We have implemented the prototype with modified Linux 2.4.18 on an ARM-based embedded system and measured the performance. Experimental results show that worst-case OS latency of a real-time process is reduced to 23% of the original one, at the expense of slowdown of non-real-time process by 20%.

Although similar performance improvement can be attained by analyzing the overall Linux kernel thoroughly and modifying a large portion of the source code, our approach is more efficient in that it modifies only a very small portion of the Linux code, and it does not require a deep knowledge of each Linux kernel function. In addition, our approach is not useful only for embedded Linux, but also for desktop Linux and all other kinds of real-time operating systems in which the critical section is significantly long.

References

[1] L. Abeni, A. Goel, C. Krasic, J. Snow and J. Walpole, "A Measurement-Based Analysis of the Real-Time Performance of Linux", In *The 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002)* , San Jose, September 2002.

[2] M. Barabanov and V. Yodaiken, "Real-time Linux", In *Linux Journal* , March 1996.

[3] A. Heursch and H. Rzehak, "Rapid Reaction Linux : Linux with low latency and high timing accuracy", In *5th Annual Linux Showcase & Conference* , Oakland, CA, November 2001.

[4] R. Love, "Interactive Kernel Performance", In *Linux Symposium* , Ottawa, Canada, July 2003.

[5] A. Heursch, D. Grambow, A. Horstkotte, H. Rzehak, "Steps Towards a Fully Preemptable Linux Kernel", In *27th IFAC/IFIP/IEEE Workshop on Real-Time Programming* , Lagow, Poland, May 2003.

[6] S. Brosky, "Shielded CPUs: Real-Time Performance in Standard Linux", In *Linux Journal* , May 2004.

[7] R. Love, The Linux kernel preemption project. http://kpreempt.sourceforge.net/.

[8] Kansas University, UTIME Patch - Microsecond Resolution Timers for Linux, http://www.ittc.ukans.edu/utime/

[9] D. Bovet and M. Cesati, Understanding Linux Kernel 2nd edition, O'REILLY, 2003.

[10] Mplayer - movie player for linux. http://www.mplayerhq.hu.

[11] QPlus embedded linux system. http://sourcefourge.net/projects/qplus/.

[12] Redhwak Linux - Real-Time Software Environment. http://www.ccur.com/.

[13] Timesys - Embedded Linux and Development Tools. http://www.timesys.com/.

[14] S3C2410 - 32bit RISC Microprocessor. http://www.samsung.com/.

[15] V. Yodaiken, The RTLinux Manifesto, Technical report, Department of Computer Science, New Maxico Institute of Technology, 1999.

[16] A. Morton, Linux scheduling latency.