# Interrupt Handler Migration and Direct Interrupt Scheduling for Rapid Scheduling of Interrupt-driven Tasks

JUPYUNG LEE and KYU HO PARK
Korea Advanced Institute of Science and Technology

In this article, we propose two techniques that aim to minimize the scheduling latency of high-priority interrupt-driven tasks, named the Interrupt Handler Migration (IHM) and Direct Interrupt Scheduling (DIS). The IHM allows the interrupt handler to be migrated from the interrupt handler thread to the corresponding target process so that additional context switch can be avoided and the cache hit ratio with respect to the data generated by the interrupt handler can be improved. In addition, the DIS allows the shortest path reserved for urgent interrupt-process pairs to be laid between the interrupt arrival and target process by dividing a series of interrupt-driven operations into nondeferrable and deferrable operations. Both the IHM and DIS can be combined in a natural way and can operate concurrently. These techniques can be applied to all kinds of interrupt handlers with no modification to them. The proposed techniques not only reduce the scheduling latency, but also resolve the interrupt-driven priority inversion problem.

We implemented a prototype in the Linux 2.6.19 kernel after adding real-time patches. Experimental results show that the scheduling latency is significantly reduced by up to 84.2% when both techniques are applied together. When the Linux OS runs on an ARM-based embedded CPU running at 200MHz, the scheduling latency can become as low as 30$\mu$s, which is much closer to the hardware-specific limitations. By lowering the scheduling latency, the limited CPU cycles can be consumed more for user-level processes and less for system-level tasks, such as interrupt handling and scheduling.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*Scheduling*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms: Measurement, Performance

Additional Key Words and Phrases: Real-time operating system, Linux, scheduling, responsiveness, latency, interrupt handling

**ACM Reference Format:**

Lee, J. and Park, K. H. 2010. Interrupt handler migration and direct interrupt scheduling for rapid scheduling of interrupt-driven tasks. ACM Trans. Embedd. Comput. Syst. 9, 4, Article 42 (March 2010), 34 pages. DOI = 10.1145/1721695.1721708 http://doi.acm.org/10.1145/1721695.1721708

Jupyung Lee is now at Samsung Electronics, Suwon, Korea.
Authors' addresses: Department of Electrical Engineering, Korea Advanced Institute of Science and Technology. (KAIST), Daejeon, Korea; email: jplee@core.kaist.ac.kr; kpark@ee.kaist.ac.kr.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2010 ACM 1539-9087/2010/03-ART42 $10.00
DOI 10.1145/1721695.1721708 http://doi.acm.org/10.1145/1721695.1721708

## 1. INTRODUCTION

As it becomes more common to run a set of complex large-scale software programs concurrently on embedded systems, the complexity of the embedded operating systems has increased to the level of PC-based operating systems. ARM-based embedded Linux 2.6 kernel source codes and Pentium-based codes, for example, are almost identical, except for a very small percentage of hardware-dependent codes. The growing functionality of embedded operating systems makes it easier to meet the growing demands of embedded system users, but this trend may adversely affect the performance of embedded systems when the limited hardware resources cannot sustain the complexity of the embedded operating systems.

OS latency, defined as the time elapsed between the arrival of an interrupt and the reactivation of the process that was waiting for the interrupt, is one of the most important performance metrics for embedded systems [Lee and Park 2005; Goel et al. 2002]. In order to support time-sensitive applications such as streaming audio/video, interactive games, and VoIP, OS latency must be kept as short as possible. In addition, it is more appropriate for the limited CPU cycles to be consumed more for user-level processes and less for system-level tasks, such as interrupt handling and scheduling, especially when the CPU is not sufficiently fast and powerful. Thus, reducing OS latency is also important for reducing the overhead of embedded operating systems.

With the recent advent of various software techniques [Lee and Park 2005; Abeni et al. 2002; Dietrich and Walker 2005; Love 2002], the OS latency of the embedded Linux operating system has been significantly reduced. The Linux OS is now fully preemptible and scheduling latency is bounded by a constant amount of time regardless of the number of processes that are running concurrently. Using the interrupt handler thread technique [Dietrich and Walker 2005; Kleiman and Eykholt 1995], the interrupt handlers can be executed not in an interrupt-context, but in a thread-context, so that the interrupt handlers are no longer treated as nonpreemptible sections. Nevertheless, considering that the OS latency is dependent on the CPU clock speed, the current level of OS latency might be unacceptable when the Linux OS runs on an embedded CPU with a relatively low clock speed. The Pentium-based Linux and ARM-based Linux can share the growing functionalities, but they might not share the level of responsiveness. Motivated by this fact, we have analyzed the Linux 2.6 kernel and real-time patches and found that there still remains additional room for further reduction in the OS latency, which can be much closer to the hardware-specific limitation, using the two proposed techniques.

Although the interrupt handler thread technique can reduce the OS latency caused by interrupt-context nonpreemptible sections, this technique causes an additional context switch per interrupt arrival, causing a longer OS latency. In this article, we propose interrupt handler migration (IHM), a novel technique to take full advantage of the interrupt handler thread technique while avoiding the additional context switch. The proposed technique can also improve the cache hit ratio with respect to the data generated by the interrupt handler and resolve the interrupt-driven priority inversion problem.

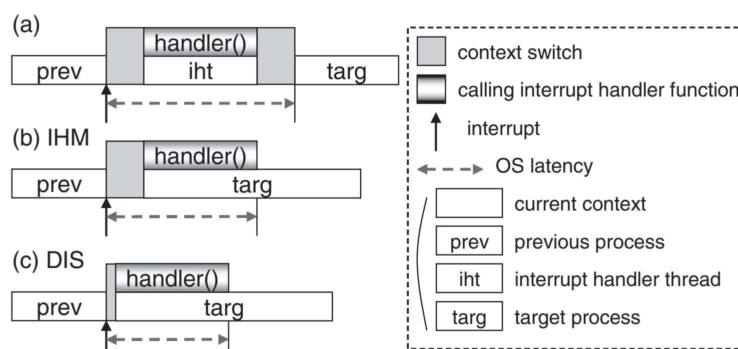



Fig. 1. Expected reduction in the OS latency that results from the proposed techniques: (a) original case, (b) when IHM is applied, and (c) when the DIS is also applied.

In addition, if we pre-process and maintain the mutual relationships between each interrupt and the target process to be reactivated by it in advance, as soon as an interrupt arrives, its urgency can be identified immediately, and if it is urgent, the next process to run can also be identified immediately. More specifically, we lay the shortest path reserved for urgent interrupt-process pairs between the interrupt arrival and the target process by dividing the series of operations that are necessary between the interrupt processing and the target process execution into nondeferrable and deferrable operations. By doing so, the OS latency can be reduced further. We have found that using this technique, even some operations that have been considered to be mandatory to activate a sleeping process, such as the *wake_up_process()* function call, can be deferred while not breaking the overall OS consistency. This technique is called direct interrupt scheduling (DIS) in this article.

Combining both the IHM technique and the DIS technique, the OS latency of high-priority processes can be significantly reduced while not sacrificing the performance of other processes and the throughput of the overall system. These techniques can be applied to all kinds of interrupt handlers.

Figure 1 illustrates the expected reduction in the OS latency that results from the proposed techniques. In summary, the IHM technique can reduce the OS latency by eliminating one context switch (Figure 1(b)) and the DIS technique can further reduce the OS latency by shortening the context switch time (Figure 1(c)). The IHM and DIS will be explained in further detail later in Sections 3 and 4, respectively.

While previous real-time patches reduce the latency caused by nonpreemptible sections, which will be detailed in the next section, our techniques reduce the latency even when an interrupt occurs in the middle of preemptible sections. That is, our techniques reduce a certain portion of the latency that cannot be reduced with previous patches. Since the worst-case latency is usually determined by nonpreemptible sections and can be reduced significantly by previous patches alone, the efficiency of our techniques is better demonstrated by observing how much closer to the hardware-specific limitation our techniques

can reduce the average latency, not by observing the worst-case latency. In addition, our techniques eliminate a set of common operations that each and every instance of the latency includes, regardless of whether it is worst case, average case, or best case. Considering these, throughout this article, our primary focus is on reducing the average latency, not the worst-case latency. However, since our techniques reduce common operations included in the latency, the worst-case latency can also be reduced as such. It is noted that reducing the average latency is beneficial not only for better support for time-sensitive applications but also for the efficient utilization of the limited CPU cyclesl, as mentioned earlier.

Considering the availability of the source code and its complexity, we chose the Linux 2.6.19 kernel running on an ARM920T core as the target OS for this study. However, the proposed techniques can also be applied to other complex operating systems.

## 2. RELATED WORK

Many studies have been conducted to improve the responsiveness of the Linux OS and other general-purpose OSs. For the Linux OS, most studies have focused on making the Linux OS fully preemptible and later reducing both the number and size of the nonpreemptible sections.

### 2.1 Evolution of Preemptibility of Linux Kernel

*Spinlock-based approach* [Love 2002] makes originally nonpreemptible Linux kernel preemptible by interpreting spinlock sections in a different perspective. The spinlock was originally introduced in Linux to support multiprocessor systems, in other words, to protect shared resources from being concurrently accessed by tasks on different CPUs [Bovet and Cesati 2003]. Since multiple tasks can be executed concurrently on a multiprocessor system as long as one does not enter a spinlock section, they can also be executed concurrently on a uniprocessor system under the same condition. This means that the Linux kernel can be considered to be fully preemptible except when a spinlock is held. In other words, a spinlock section can be interpreted as a marker that indicates a nonpreemptible section. This approach has already been added to the Linux 2.6 kernel. The problem of this approach is that locking any spinlock section prevents a higher-priority task from starting execution even when the execution poses no risk of corrupting shared data.

*Spinlock-to-mutex approach* [Dietrich and Walker 2005] converts a spinlock into a mutex in order to resolve the problem of the spinlock-based approach. It makes the kernel fully preemptible except only when a necessary mutex is already locked by other task. The limitation of this approach is that some nonpreemptible sections are protected not by spinlocks but by explicit legacy functions; thus, they cannot be converted to mutex sections. Moreover, numerous interrupt-disabled sections still remain as nonpreemptible sections. In addition, every time a high-priority task tries to hold a locked mutex, the operating causes a considerable delay including at least two context switches.

The *delayed locking technique* [Lee and Park 2005; Lee and Park 2009] seeks to overcome these limitations of the spinlock-to-mutex approach. It rearranges preemptible and nonpreemptible sections based on interrupt prediction and lock hold time prediction so that it reduces the probability that a high-priority process is blocked by low-priority processes.

In addition, *Lock-Breaking Linux* [Abeni et al. 2002] breaks long nonpreemptible sections into smaller sections. This helps the reduction of preemption delay although finding a “breakable” section and breaking it while maintaining overall OS consistency is a cumbersome, time-consuming, and sometimes logically impossible job.

All of these studies focused on increasing the probability that an interrupt occurs in the middle of a preemptible section so that a high-priority process that has been waiting for the interrupt can preempt other processes quickly. Our techniques aim to further speed up the preemption time, even when an interrupt occurs in the middle of a preemptible section.

### 2.2 Interrupt Handler Thread

In standard Linux, interrupt-context executions of both interrupt handlers and softIRQ handlers take place at priorities higher than any other processes; interrupt-context executions can always preempt any thread-context executions, and kernel preemption is disallowed during interrupt-context executions [Dietrich and Walker 2005]. Thus, both the interrupt handlers and softIRQ handlers can be seen as long nonpreemptible sections and, at the same time, the highest-priority processes. In order to convert them into preemptible sections, many developers have changed them into independent preemptible threads [Dietrich and Walker 2005; Kleiman and Eykholt 1995; Leslie et al. 1996]. By doing so, most operations previously conducted in an interrupt context can be moved to a thread context.

One problem of this technique is that it causes an additional context switch per interrupt arrival; the interrupt handler thread is interposed between the process to be preempted and the process to preempt it. Thus, it causes a longer OS latency. Our proposed technique, IHM, aims to take full advantage of the interrupt handler thread technique while avoiding the additional context switch. This is achieved by migrating the interrupt handler from the context of the interrupt handler thread to the context of the target process, that is, the process to be reactivated by the interrupt.

Lazy receiver processing (LRP) [Druschel and Banga 1996; Sundaram et al. 2000] includes a technique similar to IHM; whenever the network protocol semantics allow it, protocol processing, which is originally handled in the interrupt context, is performed lazily in the context of the target process. LRP aims to improve the fairness, stability, and throughput of a network subsystem under a high network load. LRP is fundamentally different from IHM in that the major goal is not to reduce OS latency but to increase throughput and that LRP does not consider the interrupt handler thread. In addition, while LRP focuses only on the network interrupt, IHM improves interrupt management

in general. While LRP requires the redesign of the UDP/TCP protocol stacks, IHM does not necessitate the modification of the existing Linux protocol stacks and interrupt handlers.

### 2.3 Previous Works on Interrupt Handling

Many studies have been conducted to resolve the problem that high arrival rates of network interrupts may cause the system to spend all its time processing interrupts, which is called *receive livelock* [Mogul and Ramakrishnan 1997]. This problem has been resolved by using conditional polling [Mogul and Ramakrishnan 1997; Aron and Druschel 2000; Langendoen et al. 1996; Dovrolis et al. 2001] or interrupt coalescing [Prasad et al. 2004]. Our study focuses on lowering the interrupt handling latency, called the *OS latency* in this article, under the situation where the arrival rates of the interrupts are not high enough to cause the receive livelock problem. Note that our techniques are not limited to network interrupts and are applicable to all kinds of interrupts.

There are various techniques that reduce OS latency; interrupt-triggered software prefetching [Batcher and Walker 2006] attempts to increase the cache hit ratio with respect to interrupt-driven tasks by adding prefetch instructions in the interrupt handler. If the cache locking mechanism is available, the OS latency caused by the cache miss penalty can be reduced by locking the interrupt handler codes in the cache [Puaut and Decotigny 2002]. Our proposed technique also improves the cache hit ratio with respect to the data generated by the interrupt handler, and it requires no modification of the interrupt handler.

Finally, if multiple register sets are supported by the CPU, the register save/restore procedures during the context switch can be eliminated using them [Kirsch et al. 2005], thus speeding up the context switch.

## 3. INTERRUPT HANDLING MIGRATION (IHM)

### 3.1 Basic Concept of IHM

The IHM technique basically allows the interrupt handler function to be migrated from the interrupt handler thread to the corresponding target process so that the interrupt hander function that was originally executed in the context of the interrupt handler thread can be executed in the context of the target process.

The basic concept of IHM is illustrated in Figure 2. In this figure, there are three contexts: the contexts of previous process (denoted as *prev*), interrupt handler thread (denoted as *iht*), and target process (denoted as *targ*). The *previous process* is defined as the process that has been running when an interrupt occurs, and the *target process* is defined as the process that is waiting for an interrupt in the sleep state. The *interrupt handler thread*[2] is the process that takes charge of handling the interrupt, as explained in Section 2.2.

[2]Note that in Linux, a kernel thread is equivalent to a process, except that it does not have its own address space and that it runs only in kernel mode [Bovet and Cesati 2003]. The kernel threads are treated as the same scheduling entities as user processes.

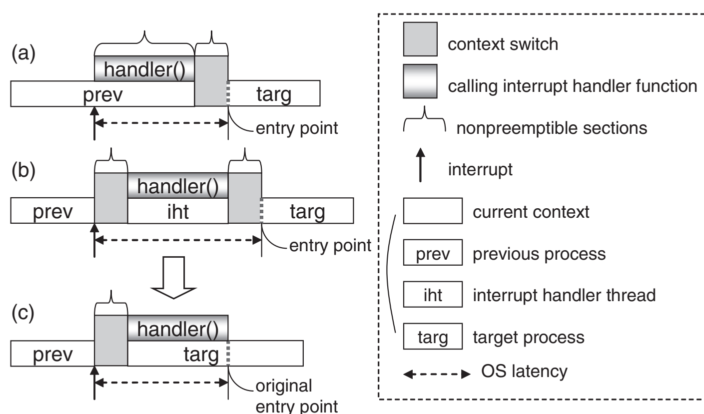



Fig. 2. Basic concept of IHM: (a) original case, (b) when interrupt handler thread is applied, and (c) when IHM is applied.

In Figure 2(a), in the original case, when an interrupt occurs, the OS suspends the currently running process, denoted as *prev*, and calls the corresponding interrupt handler function, denoted as *handler()*, in the *prev* context[3]; a context switch does not occur at this moment and the kernel becomes nonpreemptible during the execution of the interrupt handler, as explained in Section 2.2. Thus, when a more urgent interrupt occurs in the middle of the interrupt handler, it cannot be served immediately and must wait until the completion of the interrupt handler in progress.

To resolve the responsiveness problem, an interrupt handler thread is introduced, as illustrated in Figure 2(b). In this case, when an interrupt occurs, the kernel only wakes up the interrupt handler thread that corresponds to the interrupt. After the context switch[4] from *prev* to *iht*, the interrupt handler thread executes the interrupt handler function in its own context. Thus, the kernel is no longer nonpreemptible during the execution of the interrupt handler.

The problem of the interrupt handler thread is that it causes an additional context switch per interrupt arrival because the *iht* context is interposed between *prev* and *targ*, as shown in Figure 2(b). In order to avoid this additional context switch, we propose IHM. This technique was inspired by the following two observations.

— The interrupt handler function can be called not only by an interrupt handler thread, but also by any other kernel threads or processes in the kernel mode, because it is not dependent on a specific context. Thus, in the same way

[3]In this case, it is usually said that the interrupt handler is performed in the "interrupt" context to emphasize that the processes are suspended by the interrupt. However, from the perspective of the OS scheduler, it can also be said that the handler is performed in the *prev* context because no context switch has occurred at this point. Thus, the execution time consumed by the interrupt handler gets charged to the *prev* process, not the *targ* process, which is another problem resolved by the proposed technique. In this article, it will be said that the interrupt handler is performed either in the interrupt context or in the *prev* context, interchangeably.

[4]Note that the kernel also becomes nonpreemptible during a context switch.

that an interrupt handler function call is moved from an interrupt context to a thread context, it can also be moved to other thread contexts or process contexts.

—When a target process that has been waiting in the sleep state for an interrupt is reactivated by the arrival of the interrupt and is allowed to run, the starting address of the process is always after the *schedule()* function call.[5] Thus, the entry point of a sleeping target process can always be determined in compile time.

Based on these observations, we first migrate the interrupt handler function call from the interrupt handler thread to the entry point of the target process that has been waiting in the sleep state for the interrupt. We next modify the kernel further so that when the interrupt occurs, the kernel wakes up the target process, not the interrupt handler thread. We call this technique IHM and it is illustrated in Figure 2(c).

In Figure 2(c), when the interrupt occurs, the kernel wakes up the sleeping target process (*targ*). Since its entry point is the interrupt handler function, it first calls the interrupt handler function in its own context. After the completion of the handler, it enters the original entry point. In this case, an additional context switch does not occur. Furthermore, the kernel is not nonpreemptible during the execution of the interrupt handler.

In addition to avoiding the additional context switch, IHM offers the following advantages:

—*Improving the cache hit ratio*. With the IHM, since there is no address-space switch between the interrupt handler and the target process execution, the initial access to the data generated by the interrupt handler in the context of the target process does not cause a cache miss. Without IHM, the initial access causes a cold-start miss due to the address-space switch between the two. This effect will be experimentally demonstrated in Section 6.2.

—*Resolving the priority inversion problem*. If the assigned priority level of the interrupt handler thread is not well adjusted, a priority inversion problem may occur. IHM eliminates the fundamental cause of the problem. A detailed explanation will be given in Section 3.5.

—*Improving interrupt accounting*. With IHM, the execution time consumed by the interrupt handler gets charged to the corresponding target process, which is the actual consumer of the interrupt, not the interrupt handler thread or previous process. Thus, it can be said that the technique allows the kernel to account for interrupt handler costs more accurately.

Meanwhile, the introduction of IHM raises the following questions.

—*What happens if the interrupt handler does not actually wake up the process that has been predicted as the target process?* Using IHM technique, the target process is allowed to run even before the interrupt handler actually wakes it

[5]In Linux, a running process sleeps when it changes its state from TASK_RUNNING to either TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE and calls *schedule()*. Thus, the entry point of a sleeping process is next to the *schedule()* function call.

Table I. Notations for Describing the Operation of IHM

| Notation | Description |
|---|---|
| *thread* [$k$] | the process that is currently designated as the interrupt handler thread with respect to the $k$-th interrupt |
| *orig_thread* [$k$] | the process of the original interrupt handler thread with respect to the $k$-th interrupt |
| *current* | the currently running process |
| $handler(k)$ | the interrupt handler with respect to the $k$-th interrupt |

up. Thus, if the interrupt handler does not wake up the process in practice, a series of operations that are not yet allowed may be executed. To resolve this, we monitor whether the interrupt handler executed in the target process context actually wakes up the target process itself or not. If not, the target process is forced to return to the sleep state. The detailed operation will be explained in the next subsection.

— *What happens if multiple processes are waiting for the same interrupt source?* The highest-priority process among the waiting processes is allowed to be reactivated and to execute the interrupt handler upon the arrival of the interrupt. By doing so, the OS latency with respect to the highest-priority process can be reduced by IHM regardless of how many processes are concurrently waiting for the interrupt, although other sleeping processes may not take advantage of this. If the process that an interrupt handler will actually wake up can be predicted in advance, then all waiting processes can take advantage of this technique. A more detailed operation will be explained in Section 3.3.

— *What happens if a multiple number of interrupt handler threads are interposed between the previous process and target process?* For example, when a network packet is received, it is first handled by the network hardirq handler thread, then by the RX softirq handler thread, and finally by the target process. In this case, multilevel IHM is possible; that is, both handlers can be migrated to the target process in sequence. The detailed operation of this will be further explained in Section 3.4.

### 3.2 Basic Operation of IHM

In this section, the operation of IHM is explained in detail using the notations given in Table I and the flow chart shown in Figure 3. In Figure 3, the gray blocks represent the blocks added to the original procedure to support IHM.

A set of interrupt handler threads, $\{thread\,[k], 1 \leq k \leq N\}$, corresponds to each of the N interrupt sources. If IHM is not applied, the interrupt handler thread remains unchanged; thus, *thread* [$k$] is always equal to *orig_thread* [$k$]. Let us assume that the $i$-th process sleeps while waiting for the $k$-th interrupt, as shown in the figure. When the $k$-th interrupt occurs, it wakes up *thread* [$k$], that is, the original interrupt handler thread with respect to the $k$-th interrupt. If *thread* [$k$] has a higher priority than the currently running process, it preempts the process and starts to run. Then, it calls $handler(k)$, the interrupt handler function with respect to the $k$-th interrupt, in its context. The function

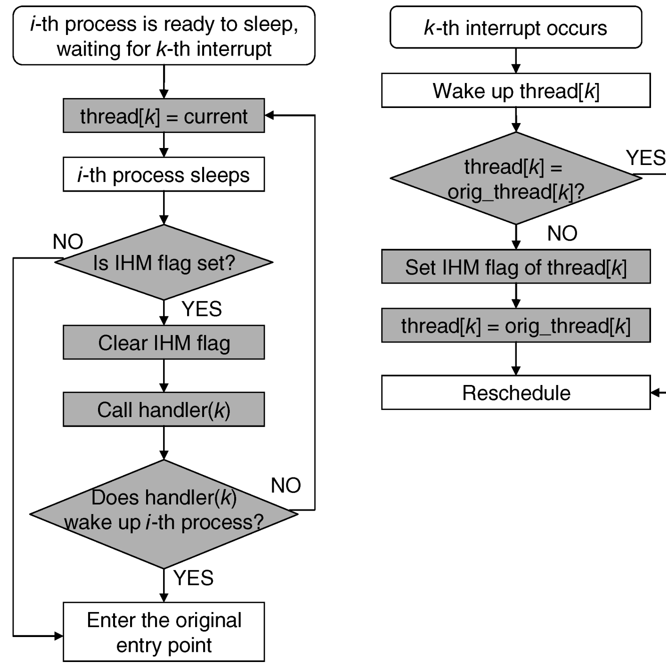



Fig. 3. The procedure of IHM; the gray-colored blocks represent blocks added to the original procedure for supporting IHM.

will then wake up the $i$-th process, which will start to run and enter the original entry point.

Conversely, if IHM is applied to the previously mentioned procedure, the gray blocks in the figure are activated. Before the $i$-th process sleeps, it changes *thread* $[k]$ from *orig_thread* $[k]$ to *current*, which is the $i$-th process itself; then, the interrupt handler thread with respect to the $k$-th interrupt becomes the $i$-th process, not the original thread. Later, when the $k$-th interrupt occurs, it wakes up *thread* $[k]$, which is not the original thread but the $i$-th process. Since *thread* $[k]$ is not equal to *orig_thread* $[k]$ at this time, the kernel can identify that IHM is ready to occur. In this case, before the kernel reschedules the processes, it first sets the IHM flag of *thread* $[k]$ to indicate that *thread* $[k]$ is activated earlier by IHM. Next, it changes the value of *thread* $[k]$ to the original value (*thread* $[k]$ = *orig_thread* $[k]$), so that the forthcoming $k$-th interrupts can be safely handled by the original interrupt handler thread as usual. If *thread* $[k]$ is equal to *orig_thread* $[k]$, it indicates that no processes are waiting for the $k$-th interrupt. In this case, the kernel immediately reschedules the processes as usual.

After these operations, the $i$-th process is allowed to run. First, it checks whether its IHM flag is set or not in order to identify whether it has been activated earlier by interrupt handler migration or normally by the interrupt handler thread. If its IHM flag is set, since *handler*$(k)$ has not been called yet, it clears the IHM flag and calls *handler*$(k)$ in the context of the $i$-th process. After that, it finally checks whether or not the execution of *handler*$(k)$ actually causes the $i$-th process to wake up; if so, the $i$-th process can safely enter the original

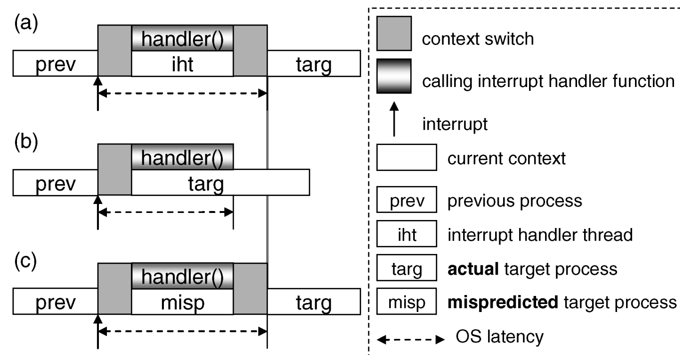



Fig. 4. Examples that show that an incorrect prediction of a target process does not increase OS latency; (a) when the IHM is not applied, (b) when the IHM is applied and the prediction of a target process is correct, and (c) when the IHM is applied and the prediction of a target process is incorrect.

entry point. In this case, it can be said that one context switch is avoided as a result of IHM. If not, the $i$-th process goes back to sleep and waits for the $i$-th process. When the IHM flag is not set, since $handler(k)$ is already called by the original interrupt handler thread, the $i$-th process can directly enter the original entry point.

Note that even when the target process that is activated earlier by IHM does not turn out to be the "actual" target process, that is, even when the target process prediction is incorrect, the incorrect prediction does not increase OS latency. This property is illustrated in Figure 4, which is an extended version of Figure 2. Figures 4(a) and 4(b) are identical to Figures 2(b) and 2(c), respectively. Figure 4(b) is the case where IHM is applied and the prediction of a target process is correct. In Figure 4(c), let us assume that a process, denoted as *misp* in the figure, is incorrectly predicted as a target process. Then, the interrupt handler is executed in the *misp* context, not in the *targ* context; thus, a context switch between *misp* and *targ* is unavoidable. Nevertheless, as shown in the figure, the OS latency with respect to the target process (Figure 4(c)) does not increase in comparison with the case where only the interrupt handler thread is applied (Figure 4(a)).

### 3.3 Operation of IHM under Multiple Waiting Processes

As mentioned in the previous section, if multiple processes are waiting for the same interrupt source in the sleep state, among them, one process should be chosen as the interrupt handler thread with respect to the interrupt source. The chosen process takes advantage of IHM; thus, one context switch can be avoided during the activation of the process. Conversely, the other waiting processes do not take advantage of the technique. Therefore, the selection principle is important for efficient use of this technique.

We basically choose the highest-priority process as the interrupt handler thread when multiple processes are waiting for the same interrupt source. By doing so, the OS latency with respect to the highest-priority process can be

reduced regardless of the number of processes that are concurrently waiting for the interrupt. Note again that the OS latency with respect to the other processes is not adversely increased, as explained in the previous section.

In order to add this selection principle to the procedure of IHM shown in Figure 3, the block denoted as "$thread\,[k] = current$" needs to be changed as follows:

$$\begin{array}{l} \text{if } (thread\,[k] = orig_thread\,[k] \text{ or } current.prio > thread\,[k].prio), \\ thread\,[k] = current \end{array} \quad (1)$$

where A.prio is the priority of process A. That is, before a process sleeps while waiting for the $k$-th interrupt, the process becomes the interrupt handler thread with respect to the $k$-th interrupt ($thread\,[k]$) only if there are no processes waiting for the $k$-th interrupt,[6] or if the priority of the process is higher than that of the current interrupt handler thread.

In addition, in the procedure shown in Figure 3, the block denoted as '$thread\,[k] = orig_thread\,[k]$' also needs to be changed as follows:

if other processes are waiting for the $k$-th interrupt,
*thread* [$k$] becomes the one with the highest priority among the processes;
otherwise, $thread\,[k] = orig_thread\,[k]$.
(2)

This means that when multiple processes are waiting for the $k$-th interrupt, the arrival of the $k$-th interrupt does not make $thread\,[k]$ equal to $orig_thread\,[k]$; instead, it makes $thread\,[k]$ equal to the waiting process with the second-highest priority so that the next arrival of the $k$-th interrupt can be handled by the process.

Conversely, if the process that an interrupt handler will actually wake up can be predicted in advance, then all waiting processes can take advantage of the IHM. In the case of a network interrupt, LRP [Druschel and Banga 1996; Sundaram et al. 2000] includes a technique that predicts which process will be activated by an incoming network interrupt before processing the interrupt, called *early demultiplexing*. LRP allows the protocol processing to be performed lazily in the context of the target process; thus, when a network packet arrives, it is required to predict which process takes responsibility for handling the packet in the early stages.

Ideally, early demultiplexing should be performed by the network interface card (NIC)[Druschel and Banga 1996]. Many commercial high-speed network adaptors contain an embedded CPU, and the necessary demultiplexing function can be performed using this CPU. In the case of inexpensive network adaptors that contain no additional hardware, however, early demultiplexing should be performed in the network interrupt handler.

In this article, we assume that the network interface card does not support early demultiplexing. Thus, if early demultiplexing is employed with the IHM, it should be performed by the network interrupt handler. To do so, we divided the original network interrupt handler into two parts: The first part handles the

[6] If $thread\,[k]$ is currently equal to $orig_thread\,[k]$, it means that no processes are waiting for the $k$-th interrupt and consequently, $thread\,[k]$ remains unchanged.

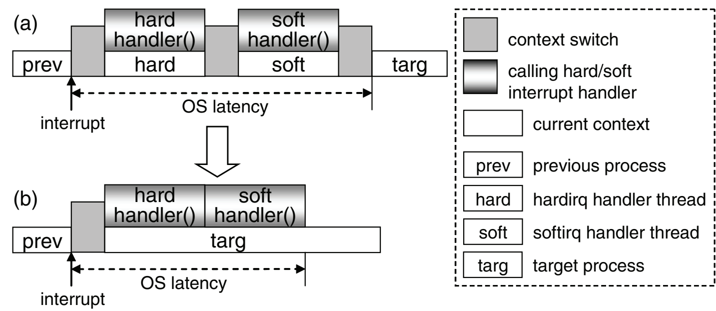



Fig. 5. The effect of IHM on multiple interposed interrupt handler threads: (a) original case, (b) when IHM is applied.

early demultiplexing and finds the target process, and the second part handles the remaining functions. The first part is performed in the interrupt context and the second part in the context of the target process. One problem that arises when early demultiplexing is used with IHM is that the kernel becomes nonpreemptible during the execution of the first part of the network interrupt handler because it is executed in the interrupt context.

In Section 6.4, we will evaluate the IHM technique both with and without early demultiplexing, focusing on the network interrupt.

Finally, note that, in general, the prediction accuracy of the target process can be further improved using a history-based approach; for example, monitoring the history of past interrupts and I/O requests from each waiting process [Zhang and West 2006] or estimating the distribution of waiting time for each process.

### 3.4 Operation of IHM under Multiple Interposed Interrupt Handler Threads

In Linux, the interrupt handlers are usually divided into hardirq handlers and softirq handlers; the former include only the critical tasks while the latter include noncritical deferrable tasks [Bovet and Cesati 2003]. Accordingly, there are two types of interrupt handler threads: hardirq and softirq handler threads. In addition, there are other event handler threads such as the tasklet, aio, and usb-storage threads. Thus, not only a hardirq handler thread but also other handler threads may be interposed between the previous process and the target process.[7] For example, when a network packet is received, it wakes up the network hardirq handler thread, net_rx softirq handler thread, and the target process consecutively. For another example, when a usb-storage interrupt occurs, it wakes up the USB hardirq handler thread, usb-storage thread, and the target process consecutively.

In this case, the IHM technique can allow multiple handlers to be consecutively migrated to the target process, which leads to a reduction of more than two context switches per interrupt. Figure 5 illustrates the effect of IHM on multiple interposed interrupt handler threads. In this example, it is assumed that both the hardirq and softirq handler threads, denoted as *hard* and *soft*,

[7]It is reminded that the definition of the previous process and the target process can be found in Section 3.2.

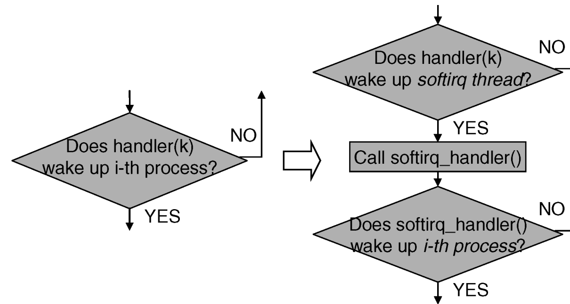



Fig. 6. A block that needs to be changed to support multilevel IHM in the entire procedure shown in Figure 3.

respectively, are interposed between *prev* and *targ*. Originally, each thread executes its own interrupt handler in each context, as shown in Figure 5(a). If IHM is applied, both handler functions can be migrated to the target process, as shown in Figure 5(b).

To support this multilevel migration, the procedure of IHM shown in Figure 3 needs to be changed slightly. Let us assume that the $i$-th process sleeps while waiting for the next network packet. Then, it is apparent that once the packet arrives and the network interrupt occurs, in the normal case, it will wake up the network hardirq thread, net_rx softirq thread, and target process. Thus, after the $i$-th process executes the hardirq handler in its context, it checks whether or not the handler actually causes the net_rx softirq thread, not the $i$-th process, to wake up. If so, it also executes the softirq handler in its context. Finally, it checks whether or not the handler causes the $i$-th process, which is itself, to wake up. If so, it can safely enter the original entry point. The modification of the procedure for IHM shown in Figure 3 is illustrated in Figure 6.

Note that the IHM technique does not require major modification of the kernel source code regarding the interrupt handler; both the hardirq and softirq handler functions are used with no modification, which is one of key advantages of this technique. In addition, if the original interrupt handler thread remains unchanged, that is, if *thread* $[k]$ is equal to *orig_thread* $[k]$, the entire interrupt handler mechanism remains unchanged. Thus, it can be said that with IHM, the interrupt handler mechanism basically remains unchanged and, if possible, it is slightly changed so that the OS latency can be reduced while the responsiveness of the overall system is not compromised.[8]

### 3.5 Resolving the Priority Inversion Problem Caused by IHM

As explained previously, with the introduction of the interrupt handler thread, the interrupt handler that was originally executed at a priority higher than any other processes is now assigned a priority that may be either higher or lower than the other processes. Due to this technique, the interrupt handler can be

[8]This means that our technique does not increase the length of nonpreemptible sections.

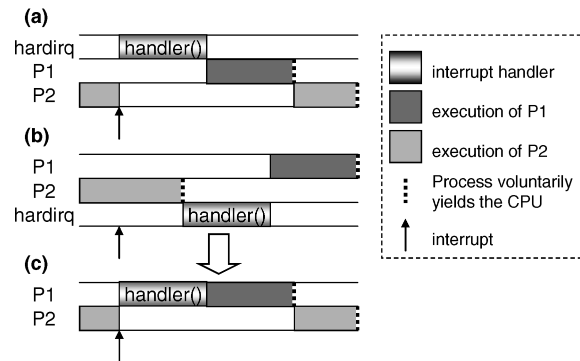



Fig. 7. Examples of the interrupt-driven priority inversion problem: (a) original case, (b) when the interrupt handler is applied, (c) when IHM is applied.

executed in a preemptible section, and thus a more urgent interrupt that occurs during the execution of other interrupt handler can be served immediately. The problem with this technique, besides the problem of causing an additional context switch, which has already been discussed in detail, is that depending on the assigned priority of the interrupt handler thread, a priority inversion problem may occur.

Figure 7 illustrates the problem. In this figure, there are two processes denoted as *P1* and *P2*, and *P1* has higher priority than *P2*. Initially, *P2* is running and the interrupt wakes up *P1*, which has been sleeping. In Figure 7(a), when the interrupt handler thread is not applied, since the interrupt handler is executed at the highest priority, *P1* can preempt *P2* as expected, and thus a priority inversion problem does not arise.

In Figure 7(b), when the interrupt handler thread is applied, it is assumed that the priority of the interrupt handler thread, denoted as *hardirq*, is lower than that of *P2*. When the interrupt occurs, since the currently running process, *P2*, has a higher priority than *hardirq*, the interrupt cannot be handled immediately and must wait until *P2* voluntarily yields the CPU. The deferred execution of the interrupt handler causes *P1* to be blocked by *P2*, which has a lower priority. In this article, we call this problem the *interrupt-driven priority inversion problem* to emphasize its cause.

Finally, in Figure 7(c), when IHM is applied, since the interrupt handler is executed in the context of *P1*, that is, the target process, it can be said that the interrupt handler is executed at the priority of the target process, not at an arbitrary priority. In this case, the priority inversion problem does not arise because the interrupt handler is executed at the priority of *P1*, which is higher than that of *P2*.

The interrupt-driven priority inversion problem basically originates from the fact that the priority level of the interrupt handler thread does not correspond to that of the target process waiting for the interrupt, although both are highly correlated. Usually, the interrupt handler thread is given a fixed priority, which is determined statically without consideration of the priorities of the other processes. The IHM technique enables the correlation of the interrupt handler and

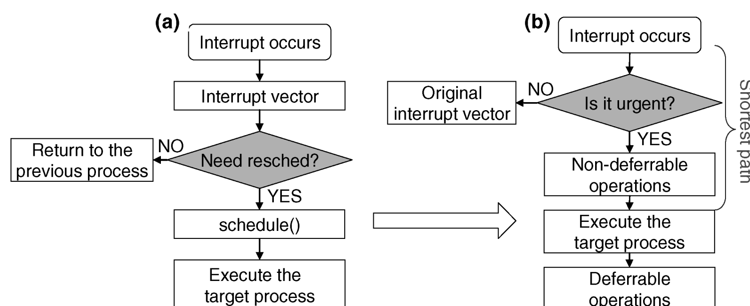



Fig. 8. Basic concept of DIS: (a) original case, (b) when DIS is applied.

the corresponding target process, and thus enables the accurate evaluation of the urgency of executing an interrupt handler.

Zhang and West [2006] also sought to resolve the interrupt-driven priority inversion problem using a simple priority-based prediction scheme in which the priority of the interrupt handler thread is set to that of the target process. This scheme shares the approach of correlating the interrupt handler thread and the target process with our technique. Unlike our technique, however, this scheme requires frequent adjustments of the priority level and does not resolve the problem of additional context switches. Our technique resolves the problems of both priority inversion and additional context switches in a unified way by migrating the interrupt handler to the target context.

## 4. DIRECT INTERRUPT SCHEDULING (DIS)

In the previous section, we showed that OS latency can be reduced using the IHM technique. In this section, we show that OS latency can be further reduced using our second proposed technique: the DIS technique.

### 4.1 Basic Concept of DIS

The basic concept of DIS is illustrated in Figure 8. First, Figure 8(a) presents a simplified illustration of the original execution path starting from the moment when an interrupt occurs. When an interrupt occurs, the CPU automatically jumps to the starting address of the interrupt service routine (ISR); the address is called the interrupt vector. Starting from the interrupt vector address, the CPU executes a series of functions related to low-level interrupt processing. During the low-level interrupt processing, if there is a target process that has been waiting for the interrupt, it is reactivated, and as an indication that rescheduling is necessary, the *NEED_RESCHED* flag is set. At the end of the low-level interrupt processing, the kernel checks whether the *NEED_RESCHED* flag is set or not, and if it is, the kernel calls the *schedule()* function to change the next process to run. If the newly activated target process has a higher priority than the currently running process, it will be chosen as the

next process to run and *schedule()* will perform a context switch between the two processes. Finally, the target process will be executed. If the *NEED_RESCHED* flag is not set, indicating that rescheduling is not necessary, the CPU simply returns to the previous process.

The basic concept of DIS originates from the observation that if the mutual relationship between each interrupt and the target process to be reactivated by the interrupt can be given in advance, as soon as an interrupt actually arrives, its urgency can be identified immediately, and if it is urgent, the next process to run can also be identified immediately. Using the preprocessed table specifying the relationship between interrupts and target processes, the urgency of an arrived interrupt can be readily identified by comparing the priority of its target process with that of the currently running process. If the interrupt turns out to be urgent, it becomes clear that the next process should be its corresponding target process. Note that such a decision can be made even before the kernel goes deeper into the low-level interrupt processing.

Once the next process to run is identified in the early stages of the interrupt processing, the next question is how the CPU can jump to the first instruction of the next process quickly in order to minimize the OS latency with respect to the next process. To do this, the operations that are interposed between the moment when an interrupt occurs and the moment when the target process starts the execution were divided into two categories: nondeferrable and deferrable operations. If the arrived interrupt is proven to be urgent and accordingly, the next process to run is identified, only nondeferrable operations are allowed at that moment so that the OS latency with respect to the target process can be minimized. In other words, the shortest path reserved for urgent interrupt-process pairs is laid between the interrupt arrival and target process so that the important interrupt-process pair processing is not delayed by less important operations. The remaining deferrable operations will be executed after the target process finishes execution. If the arrived interrupt is proven to be not urgent, the CPU jumps to the original interrupt vector as usual. We call this technique *direct interrupt scheduling* to emphasize that the rapid scheduling is directly triggered by the arrived interrupt in an early stage. The concept of this technique is illustrated in Figure 8(b).

Although DIS can be applied regardless of whether IHM is applied, it is more appropriate that both techniques are used together. When the IHM is applied, the actual interrupt handler will be executed in the context of the target process, as explained in the previous section. In this case, since the relationship between the interrupt and the corresponding target process is already managed for the IHM, the information can be reused for the DIS; thus, both techniques can be combined in a natural way. When IHM is not applied, the relationship is not available; moreover, since the CPU should first jump to the interrupt handler thread, not the target process, the efficiency of DIS is substantially degraded. Considering that DIS goes well with IHM, in the experiments section, we usually compare three cases: (i) baseline, (ii) with IHM, and (iii) with IHM and DIS.

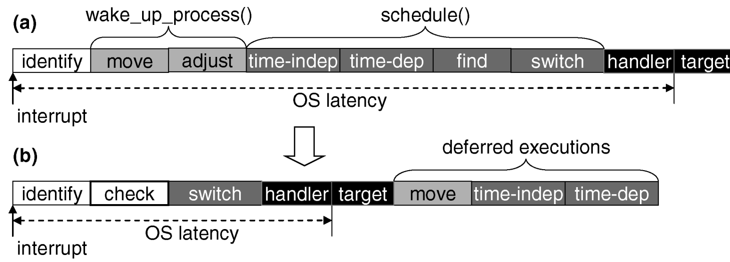



Fig. 9. The operation of DIS: (a) original case, (b) when DIS is applied. Note that the length of each component is irrelevant to actual relative execution time of each one.

Table II. The Operation of Each Functional Block Shown in Figure 9

| Name | Operation |
|---|---|
| identify | identify the arrived interrupt and the corresponding target process |
| move | move the target process from the sleep queue to the ready queue |
| adjust | adjust the priority of the newly activated target process |
| time-indep | time-independent parts of deferrable operations |
| time-dep | time-dependent parts of deferrable operations |
| find | find the next process to run among processes in the ready queue |
| switch | switch the address space, the register, and the stack |
| handler | interrupt handler corresponding to the arrived interrupt |
| target | actual execution of the target process |
| check | check the urgency of the arrived interrupt |

## 4.2 Operation of DIS

We now explain the operation of DIS in detail using the components of the operations shown in Figure 9 and Table II. It is assumed here that IHM is already applied to the operating system. Note that the sequence of the execution shown in Figure 9 is a simplified description of the actual implementation, omitting trivial components and reorganizing the order; for example, not all operations included in *time-indep* precede those included in *time-dep*. Note also that the length of each component shown in Figure 9 is irrelevant to the actual relative execution time of each component.

Figure 9(a) is the case where DIS is not applied. When an interrupt occurs, the kernel first identifies the arrived interrupt and the corresponding target process accordingly (denoted as *identify*). The next job is to wake up the target process. In Linux, the *wake_up_process()* function call is responsible for this job and includes two major operations: to move the target process from the sleep queue to the ready queue (denoted as *move*) and to adjust the priority of the target process (denoted as *adjust*).[9]

[9]The scheduler in the Linux 2.6.19 kernel, the kernel used in this study, contains an aging mechanism that increases the priority of the I/O-bound tasks and decreases the priority of the CPU-bound tasks [Bovet and Cesati 2003]. Tasks are determined to be I/O-bound or CPU-bound based on an interactivity estimator, which is based on how much time the task executes compared with how

After finishing the *wake_up_process()*, the kernel calls *schedule()* to change the next process to run. In Figure 9(a), the operations included in *schedule()* are classified into four sections: time-independent and time-dependent parts of deferrable operations (denoted as *time-indep* and *time-dep*), to find the next process to run from the processes in the ready queue (denoted as *find*), and to switch the address space, register, and stack (denoted as *switch*) between the two processes. As explained in the previous section, the operations included in *time-indep* and *time-dep* are deferrable, which include (i) recording the current time to a variable named *now*, (ii) calculating and updating time-dependent variables of the process to be preempted, including the average sleep time (*sleep_avg*),[10] the time of last process switching involving the process (*timestamp* and *last_ran*), based on *now*, (iii) updating performance statistics, and (iv) changing the states of the process to be preempted. These operations are deferred only when an arrived interrupt is proven to be urgent, in other words, when a process reactivated by an interrupt is proven to have a high priority than the current process. Furthermore, the deferred operations should be executed immediately after the reactivated process yields the CPU; otherwise, the overall OS consistency may be broken and the scheduler may not work properly. The detailed implementation issues will be presented in Section 5.2.

After finishing *schedule()*, the target process will finally start execution. Since we assumed that IHM is applied, it first calls the corresponding interrupt handler (denoted as *handler*) and subsequently enters its original entry point (denoted as *target*).

Figure 9(b) illustrates the case where DIS is applied. In this case, from the original series of operations, a set of deferrable operations are deferred when an arrived interrupt is proven to be urgent. To check the urgency of an interrupt, the operation block denoted as *check* is placed next to *identify*. The urgency is identified by comparing the priority of the target process with that of the currently running process, as explained earlier in the text. If it turns out that the interrupt is urgent, the operation blocks denoted as *move*, *time-indep*, and *time-dep* are deferred and the blocks denoted as *adjust* and *find* become unnecessary. Especially, the execution of the *time-dep* block is dependent on the current time. Thus, for the deferred execution of the *time-dep* block, the current time should be stored in a designated variable, which will be detailed in Section 5.2.

If an urgent interrupt occurs after the block denoted as *check*, it becomes clear that the target process corresponding to the interrupt will preempt the current process even before calling *wake_up_process()* and *schedule()*. Thus, all operations other than switching between the current process and target process (denoted as *switch*) can be deferred or eliminated. After the target process yields the CPU, the deferred operations should be executed immediately, as explained previously.

---

much time it sleeps [Jones 2006]. In order to estimate the interactivity in this way, before a process sleeps, it should record the time it sleeps, and after a process wakes up, it should also record the time it is awakened. Based on these time-dependent records, the priority of the target process is adjusted inside the *wake_up_process()* function call.

[10]This information is necessary in order to adjust the priority of the process according to its interactivity, as explained in the text.

One problem with this method is that since the priority of the newly activated target process may be adjusted later in the *adjust* block, the correct urgency identification may be possible only after executing the *adjust* block. This problem can be resolved using one of following solutions.

—Apply DIS to real-time processes only; in Linux, the priorities of real-time processes always remain unchanged.
—Execute the *adjust* block before the *check* block.
—Recalculate the updated priority of the target process in advance if necessary.

In this article, we use the first solution; we applied DIS to real-time processes only, which usually require better responsiveness than other non–real-time processes.

Note that if an interrupt occurs in the middle of a nonpreemptible section, since process switching is not allowed at that moment, the direct switch to the target process is delayed until the execution of the nonpreemptible section is completed.

In summary, DIS allows the shortest path between the interrupt arrival and the target process to be laid so that an important interrupt-process pair processing is not delayed by less important operations. Consequently, the OS latency with respect to the target process can be minimized. The OS latency of the Linux OS has been significantly reduced with the recent advent of various software techniques. In fact, it does not usually take long to execute a series of blocks including *move*, *adjust*, *time-indep*, *time-dep*, and *find* in a high-performance system. However, since the OS latency is dependent on the CPU clock speed, the current level of the OS latency may not be acceptable for embedded systems with relatively low CPU clock speeds. In fact, the acceleration of context switching has been achieved primarily due to increases in CPU clock speeds rather than to improvements in the efficiency of context switching itself. Conversely, our technique aims to reduce the OS latency as close to the hardware-specific limitations as possible by practical improvements in the efficiency of interrupt handling and context switching.

## 5. IMPLEMENTATION

A prototype of both IHM and DIS was implemented based on the Linux 2.6.19 kernel source code. We first added Ingo Molnar's real-time patch to the original code to support hardirq and softirq handler threads and applied the spinlock-to-mutex conversion, which reduces the OS latency caused by the nonpreemptible sections. Next, we modified the source code related to the interrupt processing and scheduling, and added a few functions.

The most recent Linux kernel that our ARM-based embedded board currently supports is 2.6.19, while at the time of the revision, the most recent kernel is 2.6.29. Although much progress in terms of reducing the OS latency has been made between 2.6.19 and 2.6.29, little of the progress is relevant to the reduction of the OS latency that the proposed techniques can bring. Specifically, in Linux 2.6.29, (i) additional context switches caused by the interrupt handler threads

```
    wait_for_network_interrupt() {
      ...
1   TRY_AGAIN:
2     if (thread[k] == orig_thread[k] ||
3         current->prio > thread[k]->prio)
4       thread[k] = current;
5     set_current_state(TASK_INTERRUPTIBLE);
6     schedule();
7     if (test_bit(IHM, current) {
8       current->ihm_target = rx_softirq;
9       hardirq_handler(k);
10    }
11    if (test_and_clear_bit(WAKEUP_EARLY, current))
12      continue;
13    else goto TRY_AGAIN;
14    current->ihm_target = current;
15    softirq_handler(rx_softirq);
16    if (test_and_clear_bit(WAKEUP_EARLY, current))
17      continue;
18    else goto TRY_AGAIN;
      /* the original entry point starts here */
    }
```

```
    __wake_up_common() {
      ...
1     if (current->ihm_target == tsk)
2       set_bit(WAKEUP_EARLY, current);
3     else
4       actual_wake_up(tsk);
    }
```

```
    irq_entry (k) {
      ...
1     wake_up_process(thread[k]);
2     if (thread[k] != orig_thread[k]) {
3         set_bit(IHM, desc->thread);
4         thread[k] = orig_thread[k];
5     }
6     schedule();
    }
```

Fig. 10. Pseudocode for IHM: the grey-colored code lines represent the original portions while the black-colored code lines represent the added portions.

still exist, (ii) the interrupt-driven priority inversion problem cannot yet be avoided, and (iii) each activation of a real-time process still includes moving it from the sleep queue to the ready queue and finding the next process to run among processes in the ready queue. That is, the same level of reduction in the OS latency is expected even when both IHM and DIS are applied to the most recent Linux kernel.

## 5.1 Pseudocode for IHM

Figure 10 shows the major modifications of the kernel source code for the IHM, omitting unnecessary parts and renaming a few code lines for easy understanding. In Figure 10, the grey code lines represent the original portions, while the black code lines represent the added portions. In Figure 10, *current* represents the process descriptor with respect to the currently running process.

The pseudocode of *irq_entry(k)*, called when the $k$-th interrupt occurs, is in accordance with the procedure of the IHM described in Figure 3. The pseudocode of *wait_for_network_interrupt()*, called when a running process decides to sleep because no network packets are received, is also in accordance with Figures 3 and 6, and Equation (1). As shown in Figure 6, the function should first check whether the hardirq_handler() wakes up the softirq thread, and next check whether the softirq_handler() wakes up the target process. To do so, *ihm_target* is added to the process descriptor, which is the process that is expected to be reactivated by the next interrupt handler. In line 8 of *wait_for_network_interrupt()*, before calling *hardirq_handler()* (line 9), *ihm_target* is set to *rx_softirq*, indicating that whether the next interrupt handler will wake up the softirq handler thread will be monitored. If *hardirq_thread()* wakes up the softirq handler thread as expected, it can be identified inside the *__wake_up_common()* function

```
   irq_entry (k) {
1    if (thread[k]->prio > current->prio) {
2      thread[k]->switch_time = current_time();
3      thread[k]->prev = current;
4      set_bit(DIS, thread[k]);
5      context_switch(current, thread[k]);
6    }
7    /* original irq_entry() functions start here */
   }
```

```
   schedule() {
1    if (test_and_clear_bit(DIS, current)) {
2      enqueue_task(current);
3      deferred_operation(current->prev, current,
4                 current->switch_time);
5    }
6    /* original schedule() functions start here */
   }
```

Fig. 11. Pseudocode for DIS.

because all kinds of wake up calls commonly call this function. Inside this function, the process that is about to wake up (*tsk*) is compared with the *ihm_target* (line 1) to find if it matches.

It is compared whether a process that is about to wake up (*tsk*) is equal to *ihm_target* (line 1). If so, the WAKEUP_EARLY bit is set as an indication that the prediction of the target process is correct (line 2). In addition, since it is not necessary to actually wake up *tsk* because *tsk* has already been awakened in advance, the wake up call is skipped. After these operations, back in the *wait_for_network_interrupt()*, if the WAKEUP_EARLY bit is set (line 11), similar operations are repeated; the *ihm_target* is set to *current* (line 14), calls the *softirq_handler()* in the current context (line 15), and checks the WAKEUP_EARLY bit (line 16). In any stage, if the WAKEUP_EARLY bit is not set, indicating that the prediction of the target process is incorrect, the function goes back to TRY_AGAIN and sleeps again (line 13 and line 18).

The key advantage of this implementation is that the modification of the kernel is minimized; wake up procedures, interrupt handlers, and the original interrupt handler threads remain unchanged except in the portions specified in Figure 10.

## 5.2 Pseudocode for DIS

Next, Figure 11 shows the major modification for the DIS. Since it is more appropriate that the DIS is applied simultaneously with the IHM, as mentioned in Section 4.1, we assume here that the IHM is already applied to the kernel source code. In *irq_entry()*, as soon as the arrived interrupt is identified, the function first compares the priority of *thread[k]*, which is currently designated as the target process with respect to the $k$-th interrupt by the IHM, with that of *current*, the currently running process (line 1). If the former is higher than the latter, the interrupt is considered to be urgent. Since the relationship between the interrupt and the corresponding target process is already managed by the IHM using a set of *thread[k]* values, it can be readily used by the DIS as shown in *irq_entry()*.

If the interrupt turns out to be urgent, the modified kernel is ready to perform the context switch between *current* and *thread[k]* even before calling *wake_up_process()* or *schedule()*. Before that, the kernel records the current time to *switch_time* (line 2) and the current process to *prev* (line 3) to execute the deferrable operations for the current process later. In addition, the kernel sets the DIS bit (line 4), indicating that the process starts to run through the

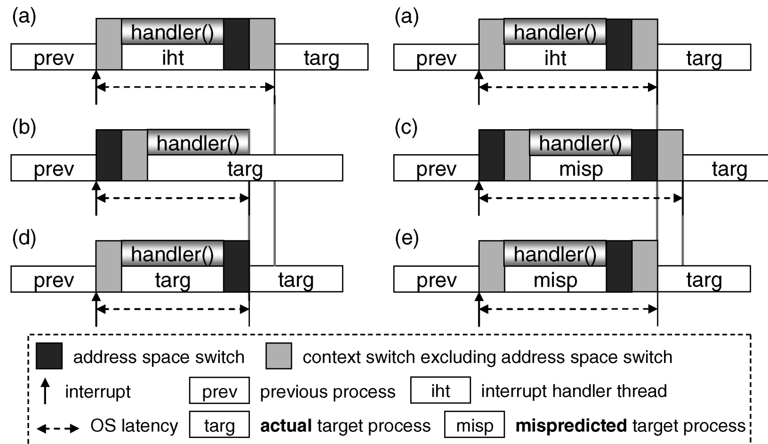



Fig. 12. Effect of an interrupt prediction of a target process when the additional overhead for address space switching is considered: (a) when the IHM is not applied, (b) when the IHM is applied and the prediction of a target process is correct, (c) when the IHM is applied and the prediction of a target process is incorrect, (d) when the LAS is applied to (b), and (e) when the LAS is applied to (c). Figure 12(a) is intentionally duplicated for vertical comparison.

shortest path due to the DIS, not through a normal path. If the interrupt is not urgent, the kernel goes through the normal path as usual (line 7).

When the process whose DIS bit is set finishes execution, the *schedule()* function will be called. In the entry point of *schedule()*, if the DIS bit of the current process is set, it indicates that the deferred operations should be executed before going further to avoid a malfunction (line 1). In this case, the current process finally enters the ready queue by calling *enqueue_task()*[11] (line 2) and the deferred operations are executed for both the previous and current processes (line 3).

### 5.3 Additional Considerations for the Implementation

5.3.1 *Deferring Address Space Switching.* Even when the prediction of the target process is incorrect, the incorrect prediction does not increase OS latency, as already explained in Section 3.2. However, if the additional overhead for address space switching is considered, the overhead may increase OS latency when the prediction is incorrect.

Figure 12 illustrates this problem. This figure is an extended version of Figure 4, dividing the context switch section into two: (i) an address space switch and (ii) the context switch excluding the address space switch. When a context switch occurs between two processes, an address space switch should also occur because each process has its own address space. However, since a kernel thread does not have its own address space and only accesses a kernel address space, when a context switch occurs between a user process and a kernel thread, an address space switch is not necessary: The kernel thread can be

[11]If the process whose DIS bit is set calls schedule() to sleep, calling *enqueue_task()* can be skipped because *dequeue_task()* will be called soon after that.

executed on the address space of the previous user-level process. This technique is called the *lazy TLB* technique in Linux [Bovet and Cesati 2003]. Since the interrupt handler thread, denoted as *iht* in Figure 12, is a kernel thread, no address space switch is necessary between *prev* and *iht*, as shown in Figure 12(a); it is only necessary between *iht* and *targ*.[12]

In Figure 12(b), when IHM is applied and the prediction of the target process is correct, one context switch between *prev* and *iht* is eliminated, and thus the OS latency is reduced. In Figure 12(c), when the IHM is applied and the prediction is incorrect, since two address space switches occur in *prev-misp* and *misp-targ*, the OS latency is increased accordingly, which is in disaccord with our previous hypothesis that an incorrect prediction does not increase OS latency.

To handle this new problem, we propose the Lazy Address Space Switching (LAS), which extends the lazy TLB technique for Linux. While a target process executes an interrupt handler that has been migrated to its own context, the execution does not require access to the address space of the target process, as is the case when the interrupt handler is executed in the interrupt handler thread. Thus, in Figures 12(b) and 12(c), the address space switch can be deferred until the moment when the target process enters its own entry point, as shown in Figures 12(d) and 12(e); these are the cases where the LAS is applied. When the LAS is applied with the IHM, the OS latency is not increased even when the prediction is incorrect.

As mentioned in Section 3.2, an additional advantage offered by IHM is that since there is no address space switch between the interrupt handler and the execution of the target process, the initial access to the data generated by the interrupt handler in the context of the target process does not cause a cache miss. When LAS is applied, however, since a deferred address space switch occurs between the two, the advantage cannot be sustained. Thus, when the prediction of the target process is sufficiently accurate, for example, when only one process is waiting for an interrupt, it is more appropriate not to apply LAS. In the experiments section, the effect of LAS on both the OS latency and the overall performance will be quantitatively evaluated.

5.3.2 *Support for SMP.* Basically, throughout this article, the target platform is assumed to be uniprocessor-based. In order to apply IHM and DIS to SMP platform, the following needs to be addressed.

—*IHM*: A race condition may occur between *wait_for_network_interrupt()* and *irq_entry()* in Figure 10: For example, if a sequence of line 1 of *irq_entry()*, line 4 of *wait_for_network_interrupt()*, and line 2 of *irq_entry()* proceeds one after another, the original thread (*orig_thread[k]*) is awakened, but IHM flag is set. The problem can be avoided simply by inserting proper spin locks between the two functions.

—*DIS*: With DIS applied, a series of deferrable operations is not executed until the target process finishes execution and *schedule()* is called. On the SMP

[12]It is assumed that the target process is not a kernel thread.

platform, *schedule()* may be called even while the target process is still running on different CPU. Thus, the pseudocode shown in Figure 11 needs to be changed; this will be our future work.

## 6. EXPERIMENTS

In this section, we present the experimental results of the performance enhancement of the Linux 2.6.19 kernel with the introduction of IHM and DIS, in terms of both the OS latency and overall performance. We first added Ingo Molnar's real-time patch [Molnar] to the vanilla kernel to enable the interrupt handler thread, which is used as as a baseline. We next modified the source code to enable IHM and DIS.

The original and modified kernel are run on an ARM-based embedded evaluation board, which includes an Intel PXA270 processor (ARM920T core) running at 200MHz, 128MB DRAM and 10Mbps ethernet transceiver. Two interrupts, timer and network interrupts, were used in the experiments as representative examples, although the proposed techniques can be applied to all kinds of interrupt sources. In the experiments involving a timer interrupt, we did not use a standard periodic timer that generates an interrupt every 10ms, but instead we used a separated hardware timer, named the *urgent timer*, which generates a timer interrupt upon request. Thus, in the experiments, when a target process sleeps while waiting for a timer interrupt, the period of the standard periodic timer remains unchanged, but instead the urgent timer is activated and its period is specified according to the requirement of the process.

The following three processes are used in the experiments.

—*Timer_task* sleeps while waiting for a timer interrupt, which is generated by the urgent timer 5ms after it sleeps. When it wakes up, it calculates the OS latency and goes back to sleep.

—*Net_task* sleeps while waiting for a network packet destined for its socket. When it wakes up, it calculates the OS latency, receives an incoming packet, and goes back to sleep.

—*Calc_task* repeats a series of arithmetic calculations only. It acts as a low-priority process disturbing the immediate scheduling of high-priority processes.

In the experiments, three cases will usually be compared: (i) baseline, (ii) with IHM, and (iii) with IHM and DIS.

### 6.1 One Process Waiting for a Timer interrupt

First, we ran the *timer_task* and *calc_task* simultaneously. *Timer_task* was given a higher priority than *calc_task*. Our goal here was to minimize the OS latency with respect to the *timer_task* in the presence of *calc_task*. The cumulative distribution function of the OS latency with respect to the *timer_task* is shown in Figure 13. The graph closer to the y-axis indicates that the OS latency is shorter.

Overall, each graph includes two separated points where a steep increase is observed. For example, in the graph denoted as *baseline*, approximately 40%

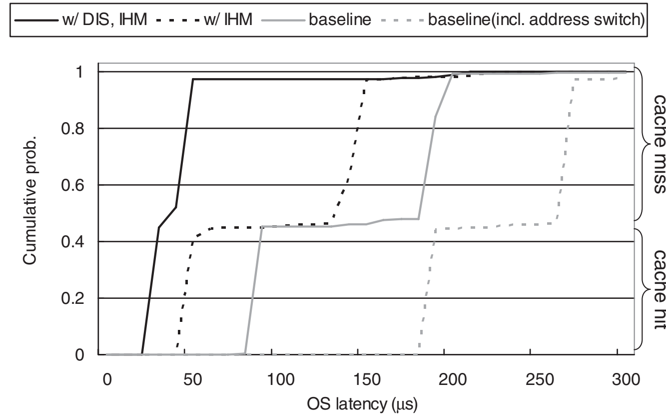



Fig. 13. The cumulative distribution of the OS latency with respect to a timer interrupt.

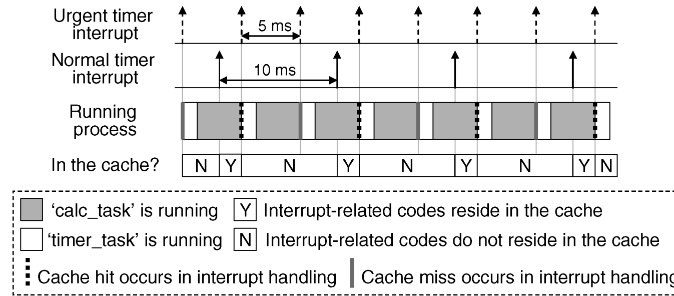



Fig. 14. An example illustrating the cache effect that affects the OS latency.

of the measured values are concentrated in approximately 80 to 90$\mu$s, while approximately 50% of them are concentrated in approximately 180 to 200$\mu$s. The separation of these two groups is a result of the cache effect; a cache hit during the execution of the interrupt handler and the schedule function leads to a low-OS-latency group, while a cache miss leads to a high-OS-latency group.

Figure 14 illustrates the cache effect. As mentioned earlier, the standard periodic timer generates a "normal" timer interrupt every 10ms while the "urgent" timer generates an urgent timer interrupt every 5ms. If an urgent timer interrupt occurs after a normal timer interrupt occurs, it usually causes a cache hit while managing the interrupt because most instructions necessary to handle the interrupt already reside in the cache. Thus, it is expected on average that one out of two urgent timer interrupts causes a cache hit during the execution of the interrupt handling and the schedule function, as illustrated in Figure 14; this is the reason why separation is observed in Figure 13.

In the graph denoted as *baseline(incl. address switch)*, the average OS latency is approximately 190$\mu$s (hit) and 270$\mu$s (miss). The latency includes the time spent switching the address space, which is accomplished by the

coprocessor in the ARM architecture. According to the analysis, the address space switching consumes a relatively long time and the consumed time cannot be avoided or reduced using the proposed technique because it is hardware-dependent. In an attempt to estimate how much closer to the hardware-specific limitations the proposed technique can reduce the OS latency, we excluded the time spent switching the address space from measuring the OS latency.

The other three cases in Figure 13 do not include the switching time. When our techniques are not applied, the average OS latency is approximately 90$\mu$s (hit) and 190$\mu$s (miss). When IHM is applied, the OS latency is reduced by 73.7% (hit) and 44.4% (miss). The one eliminated context switch per interrupt leads to this reduction. When both DIS and IHM are applied together, the OS latency is reduced by 84.2% (hit) and 81.5% (miss). This result shows that as insisted earlier, both techniques can be applied in harmony.

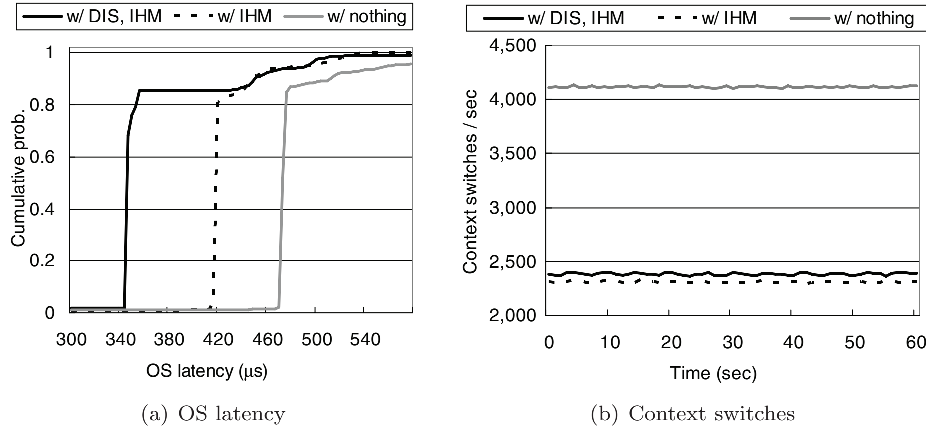
When DIS is applied in addition to IHM, the reduction in the case of a cache miss is much more outstanding. DIS reduces the number of operations necessary to make the target process, which is *timer_task* here, runnable by deferring the unnecessary operations. Thus, especially when a cache miss occurs, DIS can efficiently reduce OS latency by avoiding the cache miss penalties. When a user-level process is activated by an interrupt and is scheduled, it flushes the cache and invalidates the TLB entry; thus, it is highly likely that the handling procedure for the next interrupt causes a series of cache misses. Considering this problem, it is important to make the interrupt handling procedure as short as possible, as does DIS.

### 6.2 One Process Waiting for a Network Interrupt

Next, we ran *net_task* and *calc_task* simultaneously. The *net_task* running on an embedded board sends a request packet to a remote server, and the server sends a response packet back to the *net_task*. The round-trip time, the time taken between sending a request packet and receiving a response packet, is set to 500$\mu$s, which is long enough to allow the *net_task* to go back to sleep after sending a request packet, and short enough to prevent the *net_task* from waiting for a response packet too long. Packets are exchanged via a TCP/IP connection and the length of the response message is 10 bytes, excluding network headers.

The OS latency with respect to the *net_task* is shown in Figure 15(a). The measured values of the OS latency are much longer compared with previous results because handling a TCP/IP packet requires a much longer time than handling a timer interrupt. However, reducing the handling time itself is beyond the scope of this article. Nevertheless, the IHM still reduces the OS latency by 10.6%, and the DIS added to the IHM reduces the OS latency by 27.7%, without modifying the network interrupt handler codes. In addition, Figure 15(b) shows that the number of context switches is reduced by 43.9% due to the IHM; two context switches per response packet arrival can be eliminated using the IHM and this leads to a reduction of the OS latency.

The number of transactions per second with respect to the *net_task* is shown in Figure 16. In the experimental results, one transaction consists of sending a




(a) OS latency (b) Context switches

Fig. 15. The experimental results on one process waiting for a network interrupt.

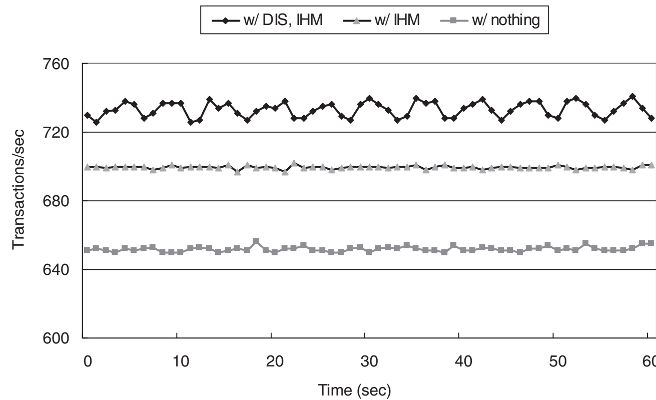
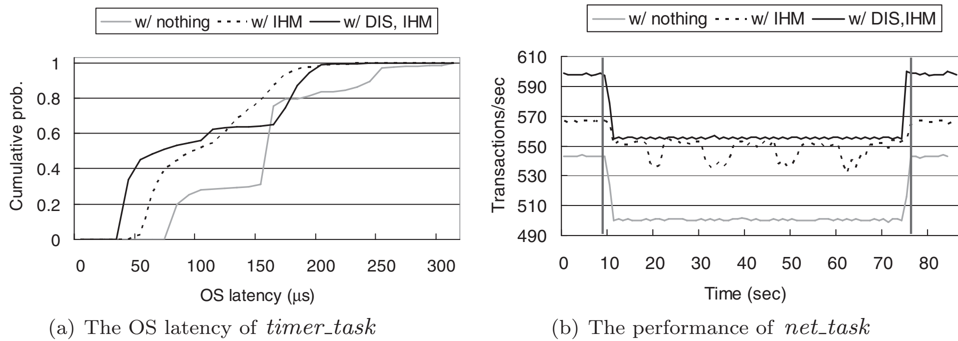



Fig. 16. The number of transactions per second; one transaction includes sending a request packet and receiving a response packet.

request packet and receiving a response packet. The result shows that the performance of the *net_task* can be improved by 7.3% and 12.5% resulting from the IHM and DIS, respectively. The improvement is not as significant as expected because the time spent executing a network interrupt handler and switching the address space, and the round-trip time remain the same regardless of whether the proposed techniques are applied. Nevertheless, IHM reduces the time spent in context switching, and DIS reduces the time spent executing "deferrable" portions of operations so that the limited CPU cycles can be consumed more for the high-priority processes and less for system-level tasks such as interrupt handling and scheduling. The more efficient utilization of the limited CPU resource leads to the performance improvement. Note that when IHM is applied, the performance improvement results from not only the lowered OS latency but also the improved cache hit ratio, as explained in Section 3.2.




(a) The OS latency of *timer_task*  (b) The performance of *net_task*

Fig. 17. The experimental results on two processes waiting for a network interrupt and a timer interrupt, respectively.

## 6.3 Two Processes Waiting for a Network Interrupt and a Timer Interrupt, Respectively

Next, we ran the *timer_task*, *net_task*, and *calc_task* simultaneously. The period of the *timer_task* is set to 9ms and the round-trip time with respect to the *net_task* is set to 700$\mu$s; both parameters were slightly increased compared with the previous experiments considering three, not two, processes are executed concurrently. Since the priority of the *timer_task* is higher than that of the *net_task*, the *net_task* may be preempted by the *timer_task* on the arrival of a timer interrupt. In addition, the priorities are specified in the order of the *timer_task*, interrupt handler threads, and *net_task*, so that the interrupt-driven priority inversion problem introduced in Section 3.5 does not arise in this experiment; if the priorities are specified in the order of the *timer_task*, *net_task*, and interrupt handler threads, the problem may arise. The problem will be discussed in the next section.

Figure 17(a) shows the OS latency with respect to the *timer_task*. Overall, compared with Figure 13, Figure 17(a) shows more irregular patterns because the *net_task* generates more complicated stress patterns than the *calc_task*, for example, more frequent access to nonpreemptible sections and more frequent generation of interrupts irrelevant to the *timer_task*. When IHM is applied, the OS latency of the *timer_task* is reduced by 30.9%. When DIS is applied together with IHM, however, DIS reduces the OS latency by a further 3.3% only. Since DIS is applied to both the *timer_task* and *net_task*, the deferrable operations of the *net_task* may be interposed between the timer interrupt and the execution of the *timer_task*, and it may increase the OS latency of the *timer_task*. As shown in Figure 17(a), DIS may increase the OS latency with a probability of approximately 40%. Nevertheless, even in the worst case, the OS latency is still lower compared with the baseline.

Figure 17(b) shows the performance of the *net_task*. In the figure, the highest-priority process (*timer_task*) is executed between the two vertical lines. Regardless of whether the *timer_task* runs concurrently or not, both DIS and IHM improve the performance of the *net_task* by 4.3% to 10.1%. When DIS is applied together with IHM, the performance fluctuation becomes considerably small.

In summary, even when multiple high-priority processes that are waiting for multiple interrupt sources run simultaneously, both IHM and DIS can efficiently improve the OS latency of these processes in a way that the limited CPU cycles are consumed more for these processes and less for system-level tasks and that a priority inversion between them does not occur.

### 6.4 Two Processes Waiting for a Network Interrupt

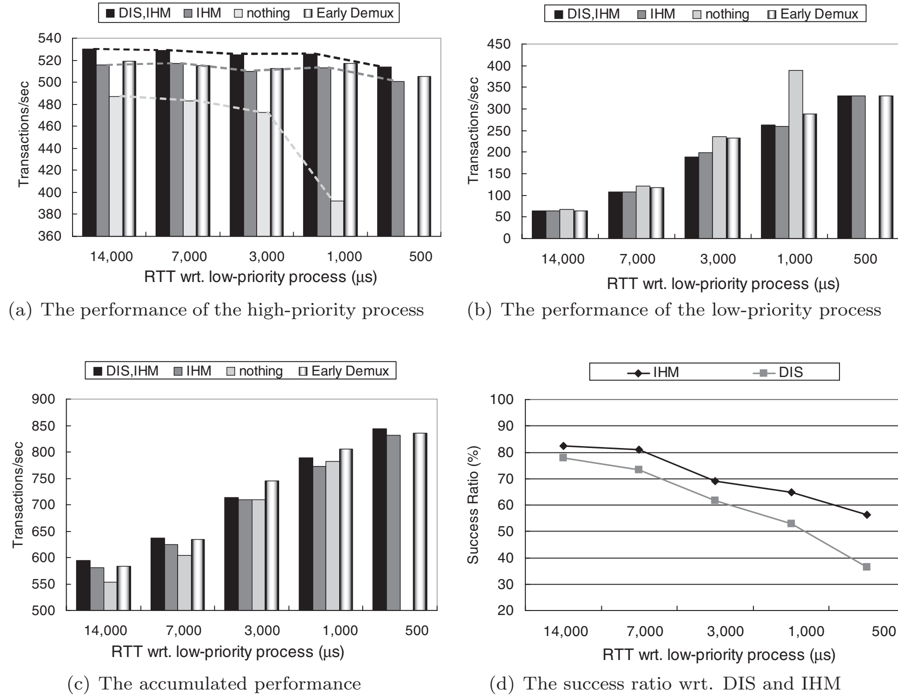
Next, we simultaneously ran two *net_task* processes with different priority levels and the *calc_task*. Each *net_task* established a TCP/IP connection to its own remote-server and exchanged request and response packets. We fixed the round-trip time with respect to the high-priority process to 1,000$\mu$s and gradually decreased the round-trip time with respect to the low-priority process from 14,000$\mu$s to 500$\mu$s, in order to gradually increase the level of stress imposed on the high-priority process. Our goal here was to maintain the performance level with respect to the high-priority process as much as possible even in the presence of a high level of stress, using DIS and IHM.

When both DIS and IHM were not applied, as the stress level increased, the performance level of the high-priority process decreased by up to 19.5%. Furthermore, especially when the round-trip time with respect to the low-priority process was 500$\mu$s, network interrupts arrived so frequently that the system spent most of the time processing interrupts, that is, the so-called receive livelock problem occurred [Mogul and Ramakrishnan 1997]. Thus, the performance could not be measured in this case.

The reason why the performance level of the high-priority process is not well-maintained is because the interrupt-driven priority inversion problem arises among the high-priority process, low-priority process, and network interrupt handler thread, as explained in Section 3.5. In this experiment, the two *net_task* processes are assigned higher priorities than the network interrupt handler thread. If a packet destined for the high-priority process arrives while the low-priority process is running, since the low-priority process has higher priority than the network interrupt handler thread, the packet cannot be served immediately and it causes the high-priority process to be blocked by the low-priority process, as illustrated in Figure 7. This problem becomes apparent in Figure 18(b): When no techniques are applied, the performance of the low-priority process is increased to an unexpected level.

When IHM is applied, the interrupt-driven priority inversion problem is resolved because it coordinates the priority level of the interrupt handler with that of the target process. Thus, the performance level of the high-priority process is well-maintained; the performance degradation does not exceed 3% even when the rate of interrupts destined for the low-priority process is double that destined for the high-priority process. In addition, compared with the baseline, the performance is increased by 5.9% to 30.9%.

When both DIS and IHM are applied, the performance level of the high-priority process is also well-maintained. Compared with the baseline, the performance is increased by 8.9% to 34.2%.




(a) The performance of the high-priority process

(b) The performance of the low-priority process

(c) The accumulated performance

(d) The success ratio wrt. DIS and IHM

Fig. 18. The experimental results on two processes waiting for a network interrupt.

As explained in Section 3.3, when multiple processes are waiting for the same interrupt source, the arrived interrupt is first handled by the highest-priority process. By doing this, the OS latency with respect to the highest priority can be maintained at the lowest level. However, the OS latency with respect to other processes may be longer. In Figure 18(b), especially when the RTT with respect to the low-priority process is 7,000$\mu$s and 3,000$\mu$s, DIS and IHM reduce the performance of the low-priority process by 11.1% to 20.2% compared with the baseline.

Figure 18(d) partially explains the reason why the performance degradation of the low-priority process arises. In the figure, the success ratio of IHM indicates the rate at which the accurate prediction of the target process occurs. Also, the success ratio of DIS indicates the rate at which the arrived network interrupt is handled directly through the shortest path. Figure 18(d) shows that both rates sharply decrease as a higher level of stress is imposed. The success ratio of IHM decreases because more network interrupts that are destined for the low-priority process occur. The success ratio of DIS decreases because a network interrupt that arrives during the execution of the high-priority process is handled by the low-priority process and the direct scheduling to the process with a lower priority than the currently running process is not allowed by DIS. The reductions of these ratios lead to the performance degradation of the low-priority process, although the performance of the high-priority process is well-maintained.

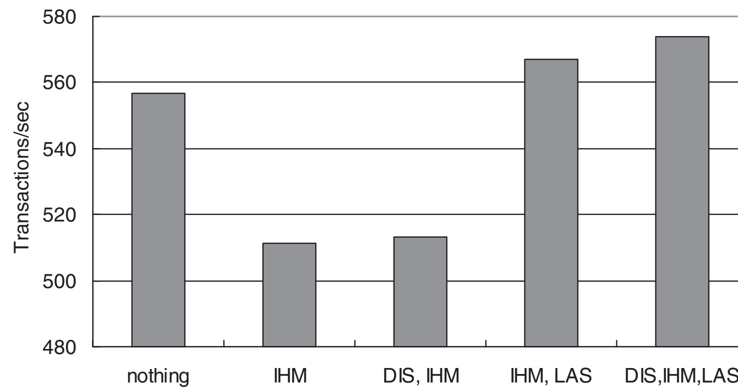



Fig. 19. The performance of the low-priority process in the worst-case situation in terms of the prediction of a target process.

If *early demultiplexing*, which is also explained in Section 3.3, is applied, the target process can be predicted in advance in the interrupt context, and thus the success ratio of the IHM can become 100%. This case is also added to Figure 18 and is denoted as *early demux*. The results show that if early demultiplexing is employed together with IHM,[13] the performance degradation of the low-priority process becomes almost negligible. Note, however, that when early demultiplexing is applied, the kernel becomes nonpreemptible during the execution of the part of the network interrupt handler because it should be executed in the interrupt context. Thus, a more urgent interrupt that occurs during early demultiplexing cannot be served immediately, as also explained in Section 3.3.

### 6.5 Effect of Misprediction of a Target Process

Finally, we repeated the previous experiment that simultaneously runs a high-priority *net_task*, low-priority *net_task*, and *calc_task* in order to quantitatively evaluate the effect of an incorrect target process prediction on the OS latency of a low-priority process in the worst-case scenario; in other words, in order to show that incorrect prediction does not increase the OS latency even in the worst case, as contended in Sections 3.2 and 5.3.1. To build the worst-case situation in terms of the target process prediction, the high-priority process is forced to idly wait idly for a response packet destined for itself which never arrives, and only the low-priority process is allowed to exchange request and response packets with a remote server at a high rate. In this situation, whenever a packet destined for the low-priority process arrives, the high-priority process is always predicted to be the corresponding target process; thus, the OS latency of the low-priority process may become longer compared with the baseline.

Figure 19 shows the performance of the low-priority process, which varies depending on the applied techniques. When IHM is applied, the performance is degraded by 7.8% to 8.1%, which is primarily due to the additional overhead

[13]In the current implementation, the early demultiplexing is not yet integrated with DIS.

for address space switching, as illustrated in Figure 12. Compared with the baseline, one address space switching is added between the previous process and the low-priority process. Conversely, when LAS, introduced in Section 5.3.1, is applied together with DIS and IHM, the performance is not degraded but slightly improved by 1.8% to 3.1%. This result is in accord with our previous hypothesis that an incorrect prediction does not increase OS latency.

## 7. CONCLUSION

In this article, we have proposed two techniques that aim to minimize the scheduling latency of high-priority interrupt-driven tasks, named *interrupt handler migration* (IHM) and *direct interrupt scheduling* (DIS). These techniques are designed based on our observation that even in highly responsive fully preemptible operating systems additional room remains for further reductions in the scheduling latency. IHM allows the interrupt handler to be migrated from the interrupt handler thread to the corresponding target process so that additional context switches can be avoided and the cache hit ratio with respect to the data generated by the interrupt handler can be improved. In addition, DIS allows the shortest path reserved for urgent interrupt-process pairs to be laid between the interrupt arrival and target process by dividing a series of interrupt-driven operations into nondeferrable and deferrable operations. The proposed techniques not only reduce the scheduling latency, but also resolve the interrupt-driven priority inversion problem by coordinating the priority level of the interrupt handler with that of the target process.

We have implemented a prototype in the Linux 2.6.19 kernel after adding real-time patches. The experimental results show that the scheduling latency is significantly reduced by up to 84.2% when both techniques are applied together. Even when the Linux OS runs on an ARM-based embedded CPU running at 200MHz, which is not fast and powerful enough to accommodate the complex modern operating system, the scheduling latency can become as low as 30$\mu$s, which is much closer to the hardware-specific limitations. By lowering the scheduling latency, the limited CPU cycles can be consumed more for user-level processes and less for system-level tasks such as interrupt handling and scheduling.

## REFERENCES

ABENI, L., GOEL, A., KRASIC, C., SNOW, J., AND WALPOLE, J. 2002. A measurement-based analysis of the real-time performance of linux. In *Proceedings of the 8th Real-Time and Embedded Technology and Applications Symposium*. IEEE, Los Alamitos, CA.

ARON, M. AND DRUSCHEL, P. 2000. Soft timers: Efficient microsecond software timer support for network processing. *ACM Trans. Comput. Syst. 18*, 3, 197–228.

BATCHER, K. W. AND WALKER, R. A. 2006. Interrupt triggered software prefetching for embedded cpu instruction cache. In *Proceedings of the 12th Real-Time and Embedded Technology and Applications Symposium*. IEEE, Los Alamitos, CA.

BOVET, D. AND CESATI, M. 2003. *Understanding Linux Kernel,* 3rd ed. O'REILLY, Sebastopol, CA.

DIETRICH, S. AND WALKER, D. 2005. The evolution of real-time linux. In *Proceedings of the 7th Real-Time Linux Workshop*.

DOVROLIS, C., THAYER, B., AND RAMANATHAN, P. 2001. Hip: Hybrid interrupt-polling for the network interface. *ACM Operating Syst. Rev*.

DRUSCHEL, P. AND BANGA, G. 1996. Lazy receiver processing: A network subsystem architecture for server systems. In *Proceedings of the 2th Symposium on Operating Systems Design and Implementation*. USENIX, Berkeley, CA.

GOEL, A., ABENI, L., KRASIC, C., SNOW, J., AND WALPOLE, J. 2002. Supporting time-sensitive applications on a commodity os. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*. USENIX, Berkeley, CA.

JONES, M. 2006. Inside the linux scheduler. *IBM developerWorks*.

KIRSCH, C., SANVIDO, M., AND HENZINGER, T. 2005. A programmable microkernel for real-time systems. In *Proceedings of the 1st International Conference on Virtual Execution Environments*. ACM, New York, 35–45.

KLEIMAN, S. AND EYKHOLT, J. 1995. Interrupts as threads. *ACM Operating Syst. Rev.*

LANGENDOEN, K., ROMEIN, J., BHOEDJANG, R., AND BAL, H. 1996. Integrating polling, interrupts, and thread management. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*. IEEE, Los Alamitos, CA.

LEE, J. AND PARK, K. 2005. Delayed locking technique for improving real-time performance of embedded Linux by prediction of timer interrupt. In *Proceedings of the 11th Real-Time and Embedded Technology and Applications Symposium*. IEEE, Los Alamitos, CA, 487–496.

LEE, J. AND PARK, K. 2009. Prediction-based micro-scheduler: Toward responsive scheduling of general-purpose operating systems. *IEEE Trans. Comput. 58*, 5, 648–661.

LESLIE, I., MCAULEY, D., BLACK, R., ROSCOE, T., BARHAM, P., EVERS, D., FAIRBAIRNS, R., AND HYDEN, E. 1996. The design and implementation of an operating system to support distributed multimedia applications. *IEEE J. Selected Areas Commun.* 14,7, 1280–1297.

LOVE, R. 2002. Lowering latency in Linux: Introducing a preemptible kernel. *Linux J*. 20, 97.

MOGUL, J. AND RAMAKRISHNAN, K. K. 1997. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst. 15*, 3, 217–252.

MOLNAR, I. Real-time preempt patch. http://people.redhat.com/mingo/realtime-preempt/.

PRASAD, R. S., JAI, M., AND DOVROLIS, C. 2004. Effects of interrupt coalescence on network measurements. In *Proceedings of the 5th Passive and Active Measurements Workshop*. Springer, Berlin, 247–256.

PUAUT, I. AND DECOTIGNY, D. 2002. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd Real-Time Systems Symposium*. IEEE, Los Alamitos, CA.

SUNDARAM, V., CHANDRA, A., GOYAL, P., SHENOY, P., SAHNI, J., AND VIN, H. 2000. Application performance in the linux multimedia operating system. In *Proceedings of the 8th Conference on Multi-Media*. ACM, New York.

ZHANG, Y. AND WEST, R. 2006. Process-aware interrupt scheduling and accounting. In *Proceedings of the 27th Real-Time Systems Symposium*. IEEE, Los Alamitos, CA, 191–201.

Received May 2008; revised February 2009; accepted April 2009