Kernel Thread Scheduling in Real-Time Linux for Wearable Computers

Dongwook Kang, Woojoong Lee, and Chanik Park

In Linux, real-time tasks are supported by separating real-time task priorities from non-real-time task priorities. However, this separation of priority ranges may not be effective when real-time tasks make the system calls that are taken care of by the kernel threads. Thus, Linux is considered a soft real-time system. Moreover, kernel threads are configured to have static priorities for throughputs. The static assignment of priorities to kernel threads causes trouble for real-time tasks when real-time tasks require kernel threads to be invoked to handle the system calls because kernel threads do not discriminate between real-time and non-real-time tasks. We present a dynamic kernel thread scheduling mechanism with weighted average priority inheritance protocol (PIP), a variation of the PIP. The scheduling algorithm assigns proper priorities to kernel threads at runtime by monitoring the activities of user-level real-time tasks. Experimental results show that the algorithms can greatly improve the unexpected execution latency of real-time tasks.

Keywords: Real-time scheduling, Linux, kernel threads, priority inheritance, wearable computers.

Dongwook Kang (phone: +82 42 860 6624, email: dkang@etri.re.kr) was with Department of Computer Science & Engineeering, POSTECH, and is currently with the Embedded Software Research Division, ETRI, Daejeon, S, Korea.

Woojoong Lee (email: wjlee@postech.ac.kr) and Chanik Park (email: chpark@postech.ac.kr) are with Department of Computer Science & Engineering, POSTECH, Pohang, S. Korea.

I. Introduction

Wearable computers have gained wide interest as a basic platform for future ubiquitous computing. We are still unsure which operating systems are good enough for the platform. Currently, Linux is regarded as the most promising alternative due to its reliability, security, and flexibility. One main disadvantage of Linux is its restricted real-time support capability.

With the advance of scheduling algorithms in the Linux kernel, Linux 2.6 provides O(1) scheduling complexity and enhances soft real-time requirements thereafter. Linux has recently gained interest in mobile terminals, automotives, robot controls, and wearable computers, which require a broad spectrum of real-time requirements. To promote Linux adoption in wearable computers, we need to further improve its real-time support capability. There has been much research conducted to improve the real-time performance of Linux. Ingo Molnar's real-time preemption patch [1] is considered the most promising method.

One of the most important features provided by Ingo Molnar's patch is the threaded interrupt [2], an interrupt handling technique in the process context rather than the interrupt context. Through the threaded interrupt technique, the preemption latency caused by interrupt handling can be reduced remarkably, resulting in more deterministic behavior of real-time tasks. However, since we need to assign priorities to each interrupt statically, the priority inversion problem [3] is inevitable. For example, in Fig. 1, two real-time tasks are related to an *IRQ thread*. In order to execute the *IRQ thread* with the highest priority among them, its priority is set to 50 while 50 and 40 are assigned to real-time tasks 1 and 2, respectively. In this situation, real-time task 1 experiences a
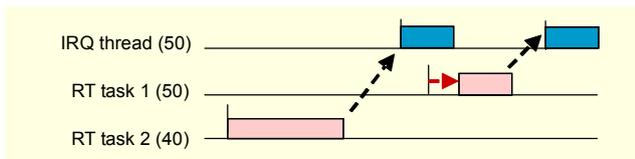
Fig. 1. The problem of IRQ thread of RT-preempt patch.

Table 1. Classification of kernel threads.

Classification		Name	Default priority
Direct group		IRQ thread	Real-time 40–50
		ksoftirqd	Real-time 1
Indirect group	System kernel thread group	pdflush	Nice 0
		kswapd	Nice 0
	Worker kernel thread group	keventd	Nice -5
		aio	Nice -5
		kthread	Nice -5
		khelper	Nice -5

considerable latency caused by the execution of the *IRQ thread*, which handles interrupts of real-time task 2, as marked by the red arrow. This is because the priority of the IRQ thread is statically assigned, whereas the related real-time tasks are changing dynamically in the current execution of the *IRQ thread*.

Linux kernel threads are special tasks which provide specific kernel services requested by both non-real-time and real-time tasks. They are different from user level application tasks in several points: they are automatically created by the kernel and are always executed at the kernel level. However, the kernel threads are treated as the same scheduling entities as user level application tasks [4].

In current Linux implementation, kernel threads are configured to have static priorities for throughputs. However, the static assignment of priorities to kernel threads causes trouble for real-time tasks when real-time tasks require kernel threads to be invoked for the kernel service via system calls. This is because kernel threads do not discriminate real-time tasks from non-real-time tasks. For example, the kernel thread called *pdflush* writes dirty memory pages back to the disk. If real-time tasks have made several I/O requests for memory page updates, and *pdflush* is not scheduled sufficiently, then the responsiveness of the real-time tasks may be prolonged significantly due to a shortage of memory cache.

In this paper, we classify the kernel threads into three groups according to how they are associated with real-time tasks. The classifications are shown in Table 1. First, kernel threads are classified into two large groups of direct and indirect groups.

Moreover, the kernel threads in the indirect group are classified into system and worker kernel thread groups.

The first classification is based on whether the response time of real-time tasks is affected by the execution delay of the kernel thread directly or indirectly. A direct group includes *IRQ thread* and *ksoftirqd* kernel threads, which wake up the corresponding real-time task at the end of interrupt handling. Thus, whenever these two kernel threads suffer an execution delay, the response time of the real-time task increases by the same amount of time as the delay.

The system kernel thread group in the indirect group includes kernel threads invoked when the current system state exceeds a system threshold configured for optimal throughputs. For example, a real-time task begins reclaiming the used memory pages instead of *kswapd* when the number of free pages becomes lower than the system threshold.

The worker kernel threads are the kernel threads serving work queues, such as *aio, kthread*, and *keventd*. They do their jobs asynchronously with real-time tasks, and thus their execution delay does not directly affect the response time of real-time tasks.

In this paper, we propose a dynamic scheduling algorithm for kernel threads in Linux, to monitor the activities of real-time tasks and assign proper priorities to kernel threads dynamically.

The remainder of this paper is organized as follows: In section II, we present related research works. The problems caused by kernel thread scheduling of current Linux implementation are exposed in section III. In section IV, we introduce our kernel thread scheduling algorithm, which solves the problems of section III. The performance evaluation of our algorithm is presented in section V. Finally, section VI presents our conclusions.

II. Related Works

There have been several research works to enhance the real-time performance of Linux [1], [2], [5]-[7]. The approaches of using a separate module [8], [9] or separate microkernel [10] are excluded from further consideration due to their less generality. We only consider the approach of real-time patches that are applied to vanilla Linux. Among the real-time patches, Ingo Molnar's patch [1], [2], [6] is considered to be promising. It introduces three techniques to enhance the real-time performance of Linux: an *IRQ thread*, RT mutex, and high-resolution timer. The *IRQ thread* is a kernel thread handling the top-halves of interrupts and is woken up by the interrupt service routines (ISRs) when interrupts occur. This technique reduces preemption latency of a vanilla kernel. It is widely used for real-time Linux, and the performance is known to be

excellent [7]. However, it requires the system administrators to configure static priorities of *IRQ threads*, and assigning static priority to an *IRQ thread* inevitably causes the priority inversion problem.

In [5], all interrupt handlers are configured to have static priorities and are executed in a process context like the *IRQ threads* of Ingo Molnar's patch. Note that in the case of [5], the entire execution of an interrupt handler is conducted in the process context, whereas in the case of [1], only some parts of an interrupt handler are conducted in the process context. However, the priority inversion problem also appears in [5] due to its static assignment of priorities to interrupt handlers.

Another real-time patch approach is MontaVista Linux [11]. This commercial extension of Linux has been enhanced to become a fully fledged real-time operating system, and various mobile phones and smart phones have been developed with it. Its core features, such as a preemptible kernel and O(1) scheduler, are adopted in mainstream Linux [11], [12].

Next, TimeSys Linux enhances real-time performance within the Linux kernel by adding mutual exclusion preemption mechanisms rather than spinlocks and improves real-time scheduling by supporting schedulable interrupts and up to 2048 process priorities. Moreover, TimeSys Linux/Real-Time, a set of loadable modules, is provided to improve timer granularity to the system clock level [13].

Finally, Linux /RK, which stands for Linux/Resource Kernel, incorporates real-time extensions to the Linux kernel to support the abstractions of a resource kernel. A resource kernel is a real-time operating system which provides timely, guaranteed, and enforced access to system resources for applications [14].

III. Scheduling Problems of Kernel Threads

Real-time tasks in Linux may suffer from increased response time when the associated kernel threads are not scheduled appropriately. In the following subsections, we describe examples of kernel threads which cause problems in the case of the current Linux kernel.

1. IRQ Thread and ksoftirqd

In Linux, top-halves of interrupts are handled in the interrupt context with a higher priority than all tasks, while the bottom-halves are handled by a kernel thread, *ksoftirqd*. However, when Ingo Molnar's real-time preemption patch is applied, top-halves of interrupts are handled by an *IRQ thread* to allow real-time tasks to preempt interrupt handlers.

To obtain the advantage to its fullest, the priorities of the *IRQ thread* and *ksoftirqd* have to be properly assigned by users. However, since the priorities are static, the priority inversion

problem may occur, which becomes more serious when the higher priority task rarely uses an external device while the lower priority task uses the device.

2. pdflush

A *pdflush* kernel thread writes back dirty pages to disks to control the dirty ratio of the system. When it is not scheduled for a long time, it can affect the response time of real-time tasks requesting disk writes or requiring additional free memory pages. To be more specific, when a real-time task performs disk read I/Os, the kernel tries to allocate multiple free pages to buffer read data. At that time, if the number of free pages is below the specified threshold due to the starvation of *pdflush*, then the real-time task reclaims used pages by itself to keep the number of free pages above the threshold. This is required because if all free memory has been used, the kernel might easily get trapped in a deadly chain of memory requests that leads to a system crash [5].

The case of a disk write task is similar. For the same reason of maintaining the dirty ratio below a specific threshold, the real-time task writes back dirty pages to the disk by itself, and its response time increases consequently when the dirty ratio crosses the threshold.

3. kswapd

A *kswpad* kernel thread swaps out the least recently used pages to maintain a number of free pages. Since this kernel thread also helps normal tasks to allocate free pages, its starvation causes the identical problems introduced in subsection III.2. For instance, when *kswapd* suffers from a long execution delay and the number of free pages enters a critical state, real-time tasks requesting additional pages have to reclaim the used pages instead of *kswapd*.

4. Worker Kernel Threads

Worker kernel threads are based on work queues. Each worker kernel thread has its own work queue, and the normal tasks insert work structures to the worker kernel threads. Then, the worker kernel thread sequentially handles the work structures in the work queue.

Execution delays of the worker kernel threads can affect the performance of real-time tasks. For example, *kthread*, a type of worker kernel thread, plays a role in creating a new kernel thread. When *pdflush* or a normal task tries to create an additional *pdflush* to help write back dirty pages, a *work* structure is inserted into the work queue of *kthread*. If the *kthread* is starved and the creation of the new *pdflush* is delayed, then the dirty ratio of the system cannot be controlled

properly. As a result, the real-time tasks can experience the same problems introduced in subsection III. 2.

As another example, an *aio* worker kernel thread performs I/O requests of normal tasks asynchronously. When a real-time task requests an I/O job from the *aio* worker kernel thread that is suffering starvation, the performance of the real-time task may decrease.

IV. Dynamic Kernel Thread Scheduling for Real-Time Systems

To solve the problems mentioned in section III, we suggest a dynamic kernel thread scheduling algorithm, weighted average priority inheritance protocol (PIP). The weighted average PIP algorithm is a variation of PIP, the priority inheritance protocol [3].

When PIP is applied to kernel thread scheduling, the priority of a kernel thread k_i is the maximum priority of real-time tasks in R_i, which is a set of real-time tasks related to k_i:

$$\text{prio}(k_i) = \begin{cases} \text{default\_prio}(k_i) & \text{if } |R_i| = 0, \\ \text{max\_prio}(R_i) & \text{if } |R_i| \neq 0. \end{cases} \quad (1)$$

In this case, the weak point is that the kernel thread can use the CPU excessively and other real-time tasks can suffer unnecessary execution latency. Originally, the kernel threads are executed in the background and perform their jobs in a manner that minimizes the interruption of normal tasks.

Thus, the alternative plan is the weighted average PIP:

$$\text{prio}(k_i) = \begin{cases} \text{default\_prio}(k_i) & \text{if } |R_i| = 0, \\ \min\left(w_i \cdot \dfrac{\sum\limits_{r_j \in R_i} \text{prio}(r_j)}{|R_i|}, \text{max\_prio}(R_i) \right) & \text{if } |R_i| \neq 0. \end{cases} \quad (2)$$

When there are no real-time tasks related with the kernel thread k_i, the default priority is assigned to k_i. Otherwise, the priority of a kernel thread is the average value of the priorities of R_i multiplied by w_i, the weight values of k_i. Since the average priority reflects the priorities of all real-time tasks associated with the kernel threads and the weight value reflects each kernel thread's own characteristics, the weighted average PIP can make up for the weak point of PIP. This priority is bounded on the maximum priority of R_i, not to disturb higher priority real-time tasks.

1. IRQ thread and ksoftirqd

To apply the weighted average PIP to an *IRQ thread* and *ksoftirqd*, the set of real-time tasks related to the two kernel threads, R_{irq}, is required. A real-time task has the relation since it requests an I/O job until the request is fulfilled by the *IRQ thread* and *ksoftirqd*. Whenever a relation is created or terminated, the priorities of the two kernel threads are recalculated using (2).

2. pdflush

A real-time task is maintained in $R_{pdflush}$ during which the dirty pages written by the real-time task exist in the page cache. In other words, the relation is started when the real-time task writes to a file yielding dirty pages, and is terminated when the dirty pages are written back to the disks.

To apply the relative importance of each real-time task to computation of the average priority, the number of files or inodes that the real-time task makes dirty is taken into consideration. For example, if a real-time task opens and writes to three different files, then the task is inserted to $R_{pdflush}$ three times. When the dirty pages in one of the three files are written back, one element of the task is removed from $R_{pdflush}$. As a result, the priority of *pdflush* becomes higher when the number of files written by higher priority tasks increases, while the priority becomes lower when the number of files written by lower priority tasks increases. This mechanism is efficient because the dirty pages are managed in the inode unit and are written back in the inode unit as well.

3. kswapd

In the case of *kswapd*, real-time tasks whose pages are in the LRU cache are the elements of R_{kswapd}. However, it is very complex to maintain the set R_{kswapd} because the number of pages in the LRU cache is in the hundreds of thousands and the pages can be shared by multiple tasks of different priorities. To simplify this problem, we assume that each real-time task has the same number of pages in the LRU cache.

In addition to w_{kswapd} and the average priority, the ratio of LRU cache pages used by all real-time tasks, *rt_page_ratio*, is used to calculate the priority of *kswapd*. The average priority is multiplied by *rt_page_ratio* to increase and decrease the priority according to the value of *rt_page_ratio*:

$$\begin{aligned} &\text{prio}(k_{kswapd}) \\ &= \begin{cases} \text{default\_prio}(k_{kswapd}) & \text{if } |R_{kswapd}| = 0, \\ \min\left(\dfrac{w_{kswapd} \sum\limits_{r_j \in R_{kswapd}} \text{prio}(r_j)}{|R_{kswapd}|} \cdot rt\_page\_ratio, \right. \\ \left. \qquad \text{max\_prio}(R_{kswapd}) \right) & \text{if } |R_{kswapd}| \neq 0. \end{cases} \end{aligned} \quad (3)$$

This prohibits increasing the priority of *kswapd* when real-time tasks use a few memory pages.

4. Worker Kernel Threads

The set of real-time tasks related to a worker kernel thread is denoted by R_{worker}. The relation begins when the real-time tasks insert a work structure to the work queue, and the relation ends when handling of the work structure is completed. Thus, whenever a new work structure is inserted or its handling is completed, the priority of the worker kernel thread is recalculated.

V. Experimental Evaluation

1. Implementation in Linux 2.6

Whenever a change occurs in R_i, a set of real-time tasks related to a kernel thread k_i, the priority of k_i is recalculated using (2). To conduct this recalculation, some information is maintained for each kernel thread, such as the default priority of k_i, R_i, current average and maximum priority of R_i, and the number of real-time tasks in R_i. Thus we add this information as member variables in TCB, which is defined as a structure *task_struct* in Linux. In the following subsections, we explain how these new variables are maintained in each kernel thread.

However, managing the set R_i is a complex routine, as R_i is needed to recalculate the maximum priority of R_i whenever a real-time task is added to or removed from it. In contrast, it is not required when obtaining the average priority of R_i. The average priority can be calculated from the previous average priority, the number of tasks in R_i, and the priority of the real-time tasks newly added to or removed from R_i. If w_i is less than 1, then the weighted average priority is always less than the maximum priority of R_i. Thus, in this case, it is not required to manage R_i. Among the kernel threads we have mentioned, *pdflush, kswapd*, and worker kernel threads belong to this case.

We implemented our new kernel thread scheduling algorithm on Linux 2.6.15 with Ingo Molnar's real-time preemption patch.

A. IRQ Thread and ksoftirqd

In subsection IV.1, we explained that a real-time task has the relation with R_{irq} since it requests an I/O job from external devices until the I/O job is completed by *IRQ thread* and *ksoftirqd*.

Thus, we add the real-time task to R_{irq} right before this task adds the I/O job to the I/O request queue and is blocked, and we remove it right after the I/O job is finished and the task is woken up.

B. pdflush

To support the dynamic scheduling of *pdflush*, a few variables are added to the inode structure, which have the number of related real-time tasks and their average priority. These are required to know the priorities of real-time tasks accessing each file.

When real-time tasks invoke a write system call to a file and pages of the file become dirty, the variables in the inode structure are updated and the priority of *pdflush* is recalculated based on the updated variables. If all dirty pages of the inode are written back to the file, then the priority of *pdflush* is also recalculated with the variables of the inode structure.

C. kswapd

As mentioned in subsection IV.3, to calculate the priority of *kswapd, rt_page_ratio* as well as the average priority of the real-time tasks are needed. To obtain *rt_page_ratio*, each page descriptor has the information of which kinds of tasks are using the corresponding page.

To record the information without additional member variables, we use a redundant bit of the *flags* member variable of the page descriptor.

D. Worker Kernel Threads

A member variable is added to the work structure, which has the priority of a real-time task that inserts the work structure into the work queue.

When work structures are inserted into work queues, the priority of the real-time task is assigned to the newly added variable and the priority of the corresponding worker kernel thread is recalculated. The priority is also recalculated when a work structure is handled completely by the worker kernel thread.

2. Performance Evaluation

We compared the performance of our implementation to that of Linux 2.6.15 which was applied with the Ingo Molnar's real-time preemption patch of version rt16 in several experiments. We used the Ubuntu 5.10 Linux distribution with NPTL 2.3.5. The run level was set to 3, where X server is not run automatically.

The experiments were performed on a machine with a 2 GHz Intel Pentium IV CPU without supporting the hyper threading technique and 256 MB DDR SDRAM. Also, an IBM IC35L080AVVA07-0 IDE disk was used.

To show the problems exposed in section III and the performance improvement made by the proposed algorithm, the experiments were conducted for kernel threads of three classifications such as *IRQ thread, ksoftirqd, pdflush, kswapd* and *kthread* worker kernel thread. The weight values were 1.2 in the case of *IRQ thread* and 0.8 otherwise. These were

selected experimentally by trial-and-error method.

A. IRQ Thread and ksoftirqd

To show the problem of static priorities of *IRQ thread* and *ksoftirqd* and how they are solved in our implementation, we measured the response time of a real-time task writing data to a file when another real-time task of lower priority reads data from another file. The first task writes 16 bytes in each period of 250 μs, while the second task reads 100 kB continuously. The two real-time tasks are related with the same *IRQ thread* and *ksoftirqd* because they use the same disk, and the first task's code snippet is presented in Fig. 2.

As shown in Figs. 3 and 4, the response time on the existing Linux with real-time preemption patch is larger than that on our implementation due to the problem explained in subsection III .1. As a result, it is proven that the problem can be solved by the weighted average PIP algorithm.

Figure 5 shows the results of the same experiment performed on Linux 2.6.18 with real-time preemption patch. This result demonstrates that the problem also occurs with the real-time preemption patch, which is enhanced with several new features

```
sleep_usec = 0; tsc1 = gettsc();
for (i = 0; i < sampling; i++) {
    tsc1 += (sleep_usec * CPU_CLOCK);
    memcpy(buf2, buf1, MEMCPY_LEN * 1024);
    write(fd, buf2, WRITE_LEN);
    tsc2 = gettsc();
    sleep_usec =
        WRITE_PERIOD - (tsc2 - tsc1) / CPU_CLOCK;
    tsc1 = gettsc();
    if (sleep_usec > 0) usleep(sleep_usec);
    else sleep_usec = 0;
}
```

Fig. 2. Essential code snippet of the disk write task used for experiments with IRQ threads.
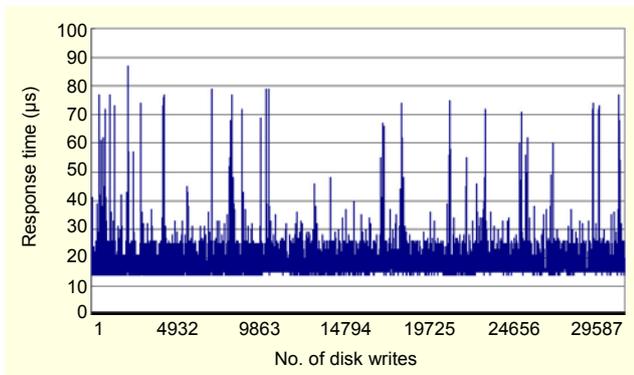
Fig. 3. Response time of the disk write task on Linux with RT-preempt patch (Linux 2.6.15).

such as a dynamic tick mechanism.

To show the effect of changing the disk read tasks' periods, we measured the response time of the disk write task, increasing the period from 0 to 200 ms.

Figure 6 shows the results. On Linux with a real-time preemption patch, the response time becomes longer as the period is increased. This is because the longer period means the more infrequent occurrence of interrupts, and disturbance from the interrupts to higher priority tasks also decreases. However, on Linux with dynamic kernel thread scheduling, changing the period does not affect the response time much. In Fig. 6, only
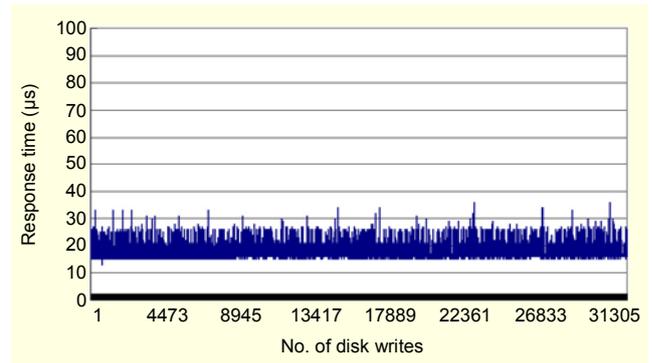
Fig. 4. Response time of the disk write task on Linux with RT-preempt patch and weighted average PIP (Linux 2.6.15).
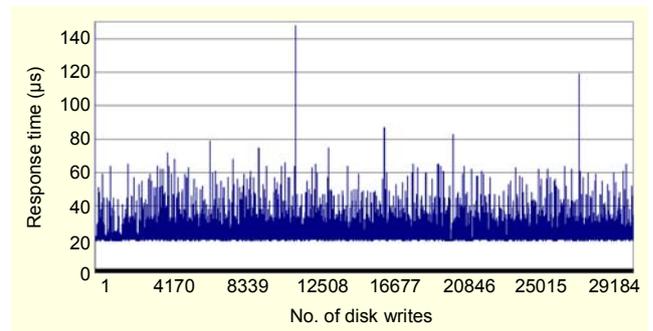
Fig. 5. Response time of the disk write task on Linux with RT-preempt patch (Linux 2.6.18).
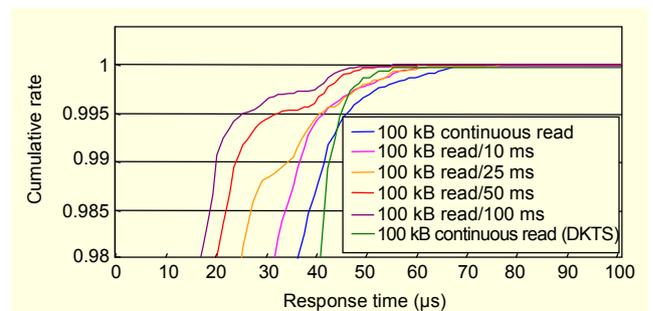
Fig. 6. Cumulative rate of response time of the disk write task on Linux with RT-preempt patch (Linux 2.6.15).

```
sleep_usec = 0; tsc1 = gettsc();
for (i = 0; i < sampling; i++) {
  tsc1 += (sleep_usec * CPU_CLOCK);
  memcpy(buf2, buf1, MEMCPY_LEN * 1024);
  write(fd, buf2, WRITE_LEN)
  tsc2 = gettsc();
  sleep_usec =
    WRITE_PERIOD - (tsc2 - tsc1) / CPU_CLOCK;
  tsc1 = gettsc();
  if (sleep_usec > 0) usleep(sleep_usec);
  else sleep_usec = 0;
}
```

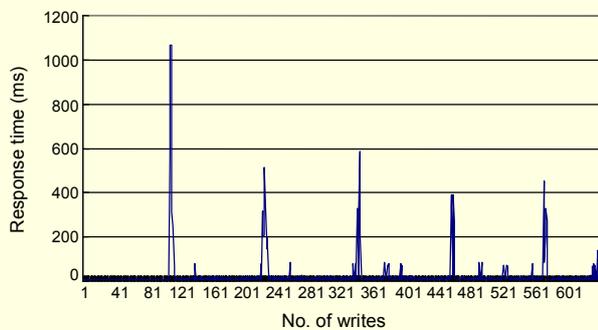Fig. 7. Essential code snippet of the disk write task used for experiments with pdflush.
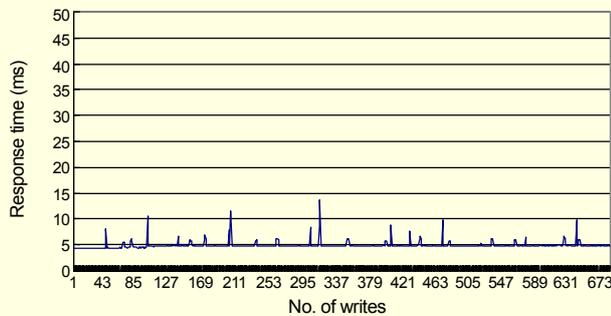

Fig. 8. Architecture of the FPGA module.

Fig. 9. Response time of the disk write task executed with a CPU-bound task on Linux with RT-preempt patch and weighted average PIP.

one case of continuous read is shown.

By comparing the results of the two kernels, the result of the new kernel crosses the results of three short periods on existing Linux, such as continuous read, 10 ms, and 25 ms. This means that the dynamic kernel thread mechanism involves additional overhead, but it stabilized the overall response time in the three cases. In the cases of periods of 50 ms and 100 ms, interrupt frequency becomes low and the disturbance to the disk write
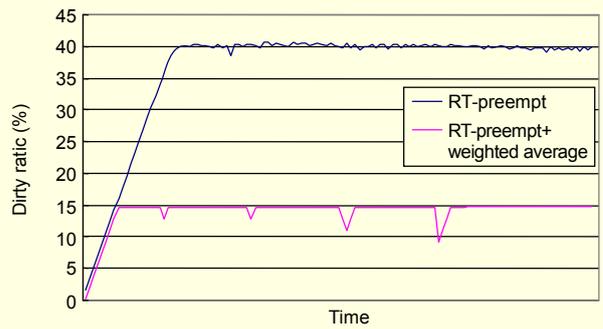

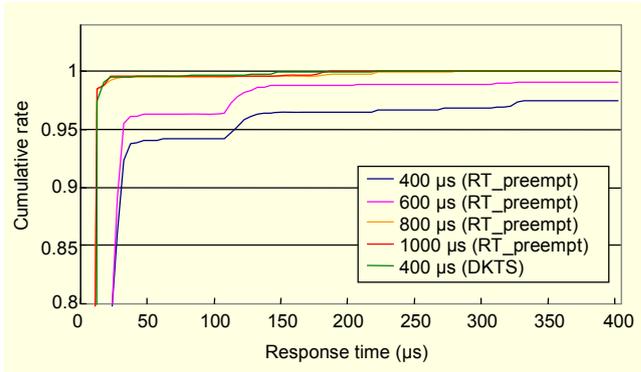Fig. 10. Dirty ratio with and without applying weighted average PIP.

Fig. 11. Cumulative rate of response time of the disk write task on Linux with RT-preempt patch (Linux 2.6.15).

task weakens. Thus, in these cases, dynamic kernel thread scheduling has few benefits.

B. pdflush

To prove the problem caused by starvation of *pdflush* and whether the proposed algorithm eliminates it to determine, we measure the response time of a real-time task writing of 800 kB to a file every 200 ms while a CPU-bound real-time task runs with lower priority. In this situation, the lower priority cpu-bound task disturbs the execution of *pdflush*. In addition to the response time, the dirty page ratio of the system is also measured. The rough source code of the disk write task is shown in Fig. 7.

Figures 8 and 9 show the response times of the two cases, respectively, and Fig. 10 shows how their dirty ratios change. The response time in the case of existing Linux with a real-time preemption patch increases from the time when the dirty ratio reaches the threshold and the real-time task writes back dirty pages by itself. In contrast, in the case of our implementation, the response time is stable since the dirty ratio is controlled below the threshold.

Figure 11 shows how changing periods of CPU-bound tasks affects the response time of the disk write task. Increasing the

```
sleep_usec = 0; tsc1 = gettsc();
for (i = 0; i < sampling; i++) {
  tsc1 += (sleep_usec * CPU_CLOCK);
  memcpy(buf2, buf1, MEMCPY_LEN * 1024);
  offset = rand() % READ_FILE_SIZE;
  lseek(fd, offset, SEEK_SET);
  read(fd, buf, READ_LEN);
  tsc2 = gettsc();
  sleep_usec =
    WRITE_PERIOD - (tsc2 - tsc1) / CPU_CLOCK;
  tsc1 = gettsc();
  if (sleep_usec > 0) usleep(sleep_usec);
  else sleep_usec = 0;
}
```

Fig. 12. Essential code snippet of the disk read task used for experiments about kswapd.
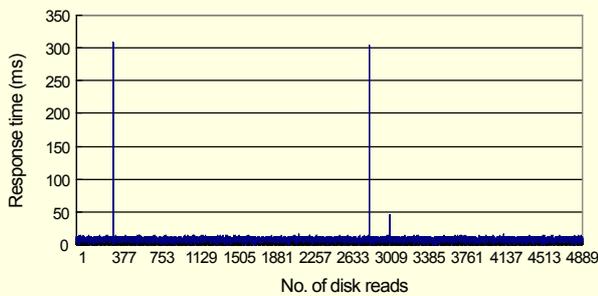

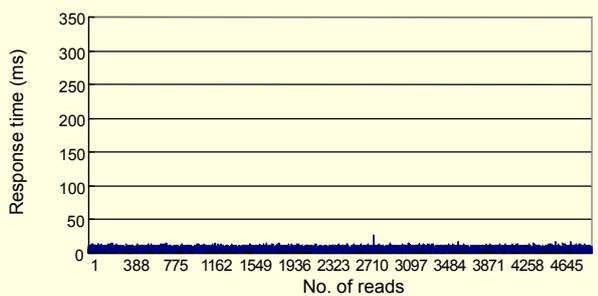Fig. 13. Response time of disk read task executed with a CPU-bound task on Linux with RT-preempt patch.

Fig. 14. Response time of disk read task executed with a CPU-bound task on Linux with RT-preempt patch and weighted average PIP.

period from 400 μs to 1 ms, we performed the same experiments. On the existing Linux, the responsibility of the disk write task improves when the period is increased and the results of the periods of 800 μs and 1000 μs are very similar to the result on the new Linux. This is because the longer period of CPU-bound tasks means less CPU consumption and less interruption to *pdflush* kernel threads.
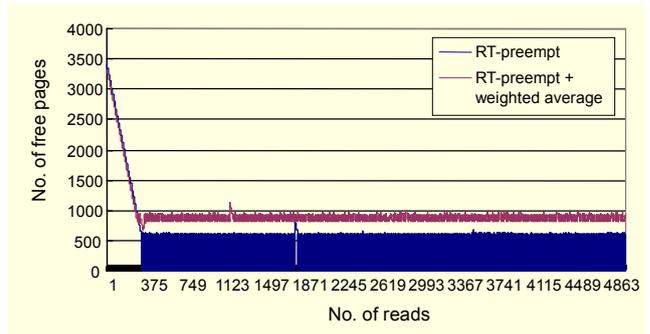

Fig. 15. The number of free pages with and without applying weighted average PIP to kswapd.
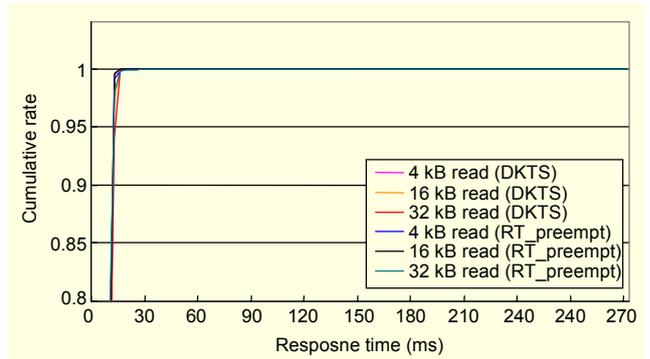

Fig. 16. Cumulative rate of response time of the disk read task on Linux with RT-preempt patch (Linux 2.6.15).

In the case of dynamic kernel thread scheduling, there is little change of response time when the period is increased. The reason is that the priority of pdflush becomes higher than the CPU-bound tasks and there is no disturbance from CPU-bound tasks. Only the result of the period of 400 μs on the new Linux is presented.

C. kswapd

Similar to the case of *pdflush*, we measured the response time of a real-time task accessing a file to read 32 kB of data every 20 ms. Along with the real-time task, multiple CPU-bound real-time tasks were executed with lower bound priorities to starve *kswapd*, and the number of free pages was also measured to monitor the behavior of *kswapd*. The snippet codes of the disk read task are shown in Fig. 12.

Figures 13 and 14 show the response times of the disk read task on the current implementation and on our implementation. Occasionally, some peaks are shown in Fig. 13, while Fig. 14 shows stable results. We confirmed that when a peak occurred, the real-time task reclaimed the used pages instead of *kswapd*.

This can also be inferred from Fig. 15. In the beginning of this figure, the numbers of free pages of both cases decrease at

the same rate. However, when reaching the points waking up *kswapd*, the number of free pages in our implementation stays larger than that of the current Linux implementation. This difference is caused by whether *kswapd* is scheduled adequately or not.

Figure 16 shows the change of the disk read tasks' responsibilities according to the read data size. For all the six cases of the two types of Linux kernel and three read data sizes, the results look similar. This is because the peak occurs on the existing Linux very infrequently as shown in Fig. 12. In fact, the peaks occurred on Linux with a real-time preemption patch for all read data sizes.

D. Worker Kernel Threads

Among the various kernel threads, the experimental target is *kthread*. As mentioned in subsection III.3, *kthread* is used to create a new *pdflush*. To show the problem introduced when *kthread* suffers starvation and creation of a new *pdflush* is delayed, we use the same experimental scenario of subsection V.2.B except that the new *pdflush* kernel threads are occasionally created by force.

In Figs. 17 and 18, the response times of the two cases are shown respectively, and Fig. 19 shows the change of their dirty
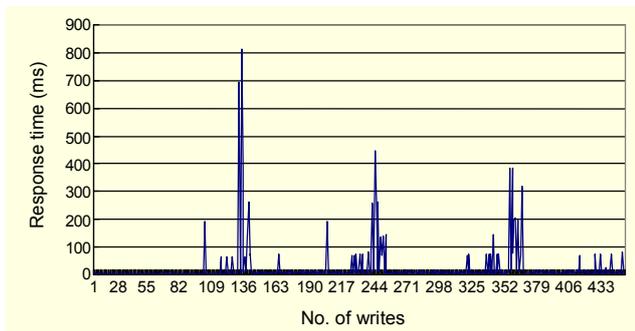

Fig. 17. Response time of the disk write task executed with kthread on Linux with RT-preempt patch.
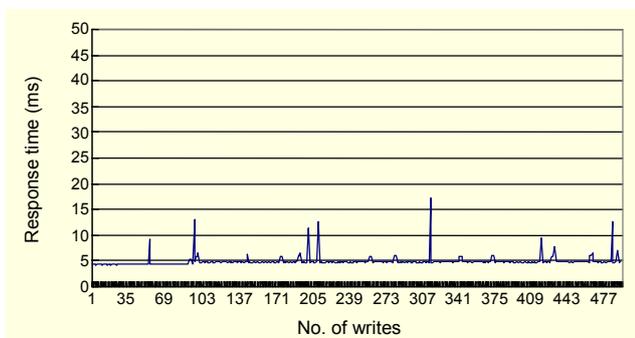

Fig. 18. Response time of the disk write task executed with kthread on Linux with RT-preempt patch and weighted average PIP.
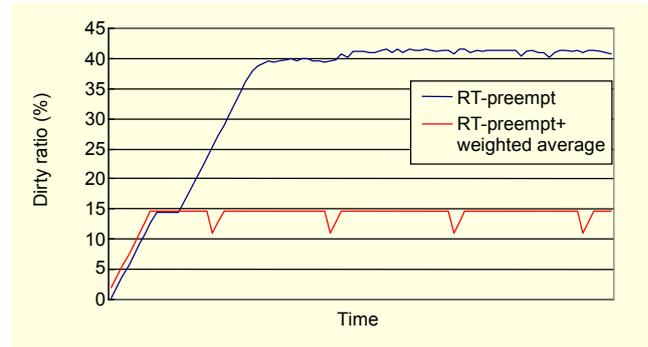

Fig. 19. Dirty ratio with and without applying weighted average PIP to kthread worker kernel thread.

ratios. In the case of the existing Linux, the dirty ratio and the response time increase because the *kthread* is not scheduled for a long time by the CPU-bound tasks and the corresponding *pdflush* waits idly until *kthread* completes the creation of the new *pdflush*. However, in our implementation, the response time and dirty ratio are stable because the priority of *kthread* becomes higher.

E. Lmbench

The following tables show the results of the lmbench benchmark tool [15] performed on Linux 2.6.15 with real-time preemption patch and Linux 2.6.15 with a real-time preemption patch and dynamic kernel thread scheduling mechanism.

Lmbench measures the latencies and bandwidths of the target system. Tables 2 to 6 show the latencies of system calls, signal handling, process creation, context switching, file creation and deletion, local communication, and memory read. Table 7 presents the bandwidth of cached file read, memory read and write, and local communications.

As shown in the tables, the system latencies and bandwidths of the two kernels do not show large differences. Consequently, the dynamic kernel thread scheduling mechanism does not

Table 2. Latencies for processor/process activities.

(μs)

	null call	null I/O	stat	open clos	selct TCP	sig inst	sig hndl	fork proc	exec proc	sh proc
rt	0.22	0.44	2.81	4.23	21	0.91	3.17	142	624	7996
dkts	0.22	0.44	2.82	4.24	21	0.91	3.19	148	627	7964

Table 3. Latencies for context switches .

(μs)

	2p/0k	2p/16k	2p/64k	8p/16k	8p/64k	16p/16k
rt	2.38	3.58	6.66	4.43	32.3	9.74
dkts	2.39	3.55	6.65	4.43	31.9	8.84

Table 4. Latencies for file accesses and VM.

(μs)

	0k file create	0k file delete	10k file create	10k file delete	mmap latency	prot fault	page fault
rt	28.9	12.4	85.4	31.2	980.0	1.312	2.0
dkts	29.1	12.4	90.8	31.2	983.0	1.309	2.0

Table 5. Latencies for local communications.

(μs)

	2p/0k	pipe	AF UNIX	UDP	TCP	TCP conn
rt	2.380	7.674	14.0	20.5	21.8	77.4
dkts	2.390	7.715	14.0	20.3	21.7	77.1

Table 6. Memory latencies.

(ns)

	L1 cache	L2 cache	Main mem
rt	0.999	9.215	116.0
dkts	0.999	9.222	116.0

Table 7. Bandwidths for local communications.

(MB/s)

	pipe	AF UNIX	TCP	file reread	mmap reread	bcopy (libc)	bcopy (hand)	mem read	mem write
rt	1448	1665	465	1313	1666.6	426.5	443.5	1658	662.4
dkts	1448	1683	478	1306	1661.9	424.1	443.5	1660	650.4

make an impact on the overall performance in general cases except those of the mentioned workloads.

VI. Conclusion

In this paper, we classified the kernel threads into three groups based on how they are associated with real-time tasks and showed how they affect real-time tasks with some examples of each kernel thread classification. Based on this information, we proposed a new scheduling algorithm for all kernel threads using a weighted average PIP mechanism. The proposed method dynamically adjusts the priorities of kernel threads by monitoring the activities of related real-time tasks. The priority computation is based on two factors: the average priority of real-time tasks related to the kernel thread and the weight value of each kernel thread representing its own characteristics.

We demonstrated by experiment that, in the case of five kernel threads, *IRQ thread, ksoftirqd, pdflush, kswapd*, and *kthread*, the response time of real-time tasks was greatly reduced when compared to the current Linux system.

References

[1] Ingo Molnar Real-time Preempt Patch, http://people.redhat.com/mingo/realtime-preempt

[2] S.T. Dietrich and D. Walker, "The Evolution of Real-Time Linux," *7th Real-Time Linux Workshop*, 2005.

[3] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. Computers*, 1990, pp. 1175-1185.

[4] R. Love, *Linux Kernel Development*, 2nd ed., Novel Press, 2005.

[5] L.E. Leyva-del-Foyo, P. Mejia-Alvarez, and D. De Niz, "Predictable Interrupt Management for Real-Time Kernels over Conventional PC Hardware," *Proc. 12th IEEE Real-Time and Embedded Technology and Applications Symposium* (RTAS'06), 2006.

[6] CE Linux Forum Realtime Preemption, http://tree.celinuxforum.org/CelfPubWiki/RealtimePreemption

[7] Patch: PREEMPT_RT and I-PIPE: The Numbers, part 4, http://lwn.net/Articles/143414/

[8] RTAI, https://www.rtai.org/

[9] RTLinux, http://www.fsmlabs.com/rtlinuxfree.html

[10] L4Linux, http://os.inf.tu-dresden.de/L4/LinuxOnL4/

[11] Montavista Real-Time Linux, http://www.mvista.com/products/realtime.html

[12] Wikipedia Montavista Linux, http://en.wikipedia.org/wiki/Montavista_Linux/

[13] Timesys Linux, http://www.realtimelinuxfoundation.org/solutions/solutions.html#SOLUTIONS_TIMESYS

[14] Linux /RK, http://www.cs.cmu.edu/~rajkumar/linux-rk.html

[15] Lmbench, http://www.bitmover.com/lmbench/

[16] D.P. Bovet and M. Cesati, *Understanding the Linux Kernel*, O'Reilly, 2005.

[17] REAL-TIME LINUX BENCHMARKS, http://www.mvista.com/products/realtime_benchmarks.html

[18] J. Mehaffey, "MontaVista Linux Open Source Real Time Project (White Paper)," MontaVista Software, 2004.

Dongwook Kang received the BS degree in computer engineering from Dongguk University, Korea, in 2004. He also received the MS degree from the Graduate School for Information Technology of POSTECH, Korea, in 2007. After graduation, he joined Electronics and Telecommunications Research Institute, Daejeon, Korea. His research interests include real-time systems and embedded systems.

Woojoong Lee received the BE degree in chemical engineering and the MS degree in computer science from Hanyang University, Korea, in 2002 and 2004, respectively. He is currently a PhD candidate in the Department of Computer Science and Engineering, POSTECH, Korea. His research interests include pervasive computing, storage systems, and embedded systems.

Chanik Park received the BE degree from Seoul National University, Seoul, Korea, in 1983, and the MS and PhD degrees from Korea Advanced Institute of Science and Technology, Korea, in 1985 and 1988, respectively. Since 1989, he has been working for POSTECH, where he is currently a professor in the Department of Computer Science and Engineering. He was a visiting scholar with the Parallel Systems group in the IBM Thomas J. Watson Research Center in 1991. He was also a visiting professor with the Storage Systems group in the IBM Almaden Research Center in 1999. He has contributed to a number of international conferences, serving as a program committee member. His research interests include storage systems, embedded systems, and pervasive computing.