

Implementing Hybrid Operating Systems with Two-Level Hardware Interrupts *

Miao Liu¹, Zili Shao², Meng Wang², Hongxing Wei¹, Tianmiao Wang¹
The Robot Research Institute¹ Department of Computing²

Beihang University

Beijing 100083, China

{wtm, threewater, whx}@me.buaa.edu.cn

The Hong Kong Polytechnic University

Hung Hom, Kowloon, Hong Kong

cszshao@comp.polyu.edu.hk

Abstract

In this paper, we propose to implement hybrid operating systems based on two-level hardware interrupts. To separate real-time and non-real-time hardware interrupts by hardware, we show that it is easier to build up hybrid systems with better performance. We analyze and discuss the key issues for implementing a hybrid system based on this and implement a hybrid system called RTLinux-THIN (Real-Time LINUX with Two-level Hardware INterrupts) on the ARM architecture by combining ARM Linux kernel 2.6.9 and μ C/OS-II. We conduct experiments on a set of real application programs including mplayer [20], Bonnie [4] and iperf [13] and compare the interrupt latency distributions for RTLinux-THIN (with and without cache locking), RTAI and Linux on a hardware platform based on Intel PXA270 processor [12]. The experimental results show that RTLinux-THIN improves real-time interrupt latencies and provides better predictability.

1 Introduction

Combining both a real-time and a time-sharing subsystem, hybrid operating systems can provide both predictable real-time task execution and non-real-time services with well-known interfaces and lots of existing applications. In order to achieve relatively low development and maintenance costs, the time-sharing subsystem of a hybrid system is often based on commodity operating systems, such as Linux [18,21,23,24,27,30]. Since commodity operating systems usually focus on general-purpose computing, it becomes an important problem how to effectively use them in real-time environments without impairing the predictability of real-time applications.

To solve the predictability problem in hybrid systems, many techniques have been proposed from the previous work. A general approach used in these techniques is to defer non-real-time tasks when there are real-time tasks

awaiting services by modifying the interrupt-handling code of a commodity O.S. [31]. Based on this, various techniques have been proposed with different methods such as hierarchical scheduling [7–11, 14, 15, 17, 19, 25], static cache locking [1, 6, 22, 28], and dynamic cache locking [2, 3, 5], etc.

Most of above work is based on one-level hardware interrupts, in which both real-time and non-real-time interrupts are coming from the same interrupt request entry and they will be separated in the interrupt handling code. We found that some problems are caused for hybrid systems by the one-level hardware interrupts. We use RTAI, an open-source hybrid system based on Linux, as the representative for commodity-O.S.-based hybrid systems, and discuss these problems below. We choose RTAI because it is being actively developed and supported.

In RTAI, the Linux O.S. kernel is treated as the idle task, and it only executes when there are no real-time tasks to run and the real-time kernel is inactive. The interrupt-handling code is modified to emulate the function of interrupt controller, so the Linux task can never block real-time interrupts or prevent itself from being preempted [18] (See Section 2 for details). There are several problems by using the interrupt-handling code to separate real-time and non-real-time interrupts and emulate interrupt controller as shown in the following:

- Interrupt disabling is frequently used in interrupt handlers, critical sections, and so on, in the Linux task. However, the interrupt disabling is processed by setting a flag in the interrupt-handling code based on the software emulation of interrupt controller without really disabling interrupts in order to avoid blocking real-time interrupts in RTAI. Therefore, non-real-time interrupts of the Linux task can still be responded when they should be disabled during interrupt disabling. Although these interrupts will be turned off later if the flag is set in the interrupt-handling code, they can only be turned off individually. These unnecessary interrupt responses and processing not only degrade the performance of the Linux task but also in-

*This work is partially supported by HK POLYU A-PH13, A-PA5X and A-PH41, Hong Kong. RTLinux-THIN can be downloaded from: <http://www4.comp.polyu.edu.hk/~cszshao/software/RT-THIN>.

crease the unpredictability of the real-time subsystem.

- The size of the interrupt-handling code for interrupt requests is increased with the functions of identification and emulation. With the big code size, it is difficult to lock the interrupt-handling code into the cache, and various cache locking techniques [2,3] may not be applied to improve the predictability.
- Although a hardware abstraction layer (HAL) is abstracted in RTAI [18], the codes related to interrupts in the Linux O.S. kernel must be rewritten. This causes a lot of work to port a hybrid system on various processors.

To solve these problems, we propose to implement hybrid operating systems based on two-level hardware interrupts. To separate real-time and non-real-time hardware interrupts by hardware, we show that it is easier to build up hybrid systems with better performance. Our focus is on improving the predictability and real-time interrupt latency of the real-time subsystem, and enhancing the performance of the time-sharing subsystem as well. Two-level hardware interrupts with different interrupt request entries have been provided in high-end embedded processors such as those based on the ARM architecture [26]. With this architecture support, our scheme not only provides an easy method for implementing hybrid systems but also achieve the performance improvement for both the time-sharing and real-time subsystems. Our main contributions are summarized as follows:

- We analyze and discuss the key issues for implementing a hybrid system based on two-level hardware interrupts including the methodology of combining the real-time kernel and the timing-sharing O.S. kernel, the implementation of the real-time scheduling and the analysis of real-time interrupt latency.
- We implement a hybrid system called RTLinux-THIN (Real-Time LINUX with Two-level Hardware INterrupts) on the ARM architecture by combining ARM Linux kernel 2.6.9 and $\mu\text{C}/\text{OS-II}$, two widely-used kernels with source code available in embedded and real-time fields (ARM Linux for time-sharing systems and $\mu\text{C}/\text{OS-II}$ for real-time systems).
- We implement our RTLinux-THIN on a hardware platform based on Intel PXA270 processor [12]. We conduct experiments with three real application programs: mplayer [20], Bonnie [4] and iperf [13] and compare statistical interrupt latency distributions in four system conditions: idle, intensive memory access (decoding a fragment of MPEG-4 video in mplayer), intensive IDE Disk access (evaluating the speed of the filesystem with a set of file operating benchmarks in Bonnie) and intensive network communication (measuring TCP/UDP bandwidth performance in iperf). The experimental results show that RTLinux-THIN

improves real-time interrupt latencies and provides better predictability.

The remainder of the paper is organized as follows: In Section 2, we present the necessary background by introducing basic knowledge of interrupts and analyzing the interrupt processing and worst-case interrupt latency of RTAI. In Section 3, we discuss the key issues of implementing a hybrid system based on two-level hardware interrupts. The system implementation and the experiments are shown in Sections 4 and 5, respectively. In Section 6, we conclude the paper.

2 Background

In this section, we present the background for interrupts and interrupt handling in hybrid systems as interrupt handling plays a very important role in implementing commodity-O.S.-based hybrid systems. We first introduce interrupts, interrupt handlers and interrupt latencies in Section 2.1. Then we analyze interrupt handling and the worst case real-time interrupt latencies in RTAI in Section 2.2.

2.1 Interrupts in Hybrid Systems

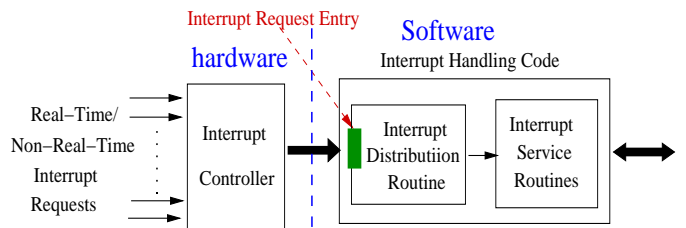


Figure 1. A typical interrupt request handling procedure in a hybrid system.

Figure 1 shows a typical interrupt request handling procedure in a hybrid system. Basically, real-time and non-real-time interrupt requests are passed to the interrupt handling code through the interrupt request entry. The interrupt handling code can be separated into two parts, the interrupt distribution routine and interrupt service routine (ISR). First, the interrupt distribution routine determines the entry point for an interrupt request. Next, a specific interrupt service routine is called, and then the interrupted task/program/interrupt is resumed or a new task/interrupt is rescheduled to be executed before we exit from the ISR.

Although the interrupt handling in hybrid systems looks like that in general-purpose O.S, it has to be changed a lot with the structure shown in Figure 1, in which real-time and non-real-time interrupts are passed through the same interrupt request entry. First, in the interrupt distribution code, in order to satisfy the predictability of the real-time subsystem, we need to separate real-time and non-real-time interrupts since they will be processed differently. Second, we have to solve the interrupt disabling problem when dealing with non-real-time interrupts.

The interrupt disabling problem is caused by that the timing-sharing subsystem of a hybrid system is usually

treated as the task with the lowest priority. With the lowest priority, the timing-sharing subsystem task can not block real-time interrupts or can not prevent itself from being preempted. On the other hand, in a timing-sharing operating system, such as Linux, interrupt disabling is frequently used in interrupt handlers, critical sections, and so on. And in most processors, interrupt disabling is achieved by masking the interrupt disabling/enabling bit in the PSW (Program Status Word) register, and all interrupt requests will be disabled if the bit is set. Therefore, in hybrid systems, we can not really set the interrupt disabling/enabling bit for interrupt disabling from the timing-sharing subsystem task. Currently, a general approach to solve this problem is to use software to emulate such interrupt control [18, 30]. Next, we will use RTAI as an example to analyze the detailed interrupt handling and worst-case real-time interrupt latency.

2.2 Interrupt handling in RTAI

RTAI is a Linux-based hybrid system, in which the Linux kernel is treated as the idle task, and it only executes when there are no real-time tasks to run and the real-time kernel is inactive [18]. In RTAI, to solve the interrupt disabling problem, in the interrupt distribution routine, a software emulation method called virtual interrupt controller based on adaptive domain environment for operating systems [29] is used to manage the interrupts of the Linux task. Basically, when the Linux task disables interrupts, the interrupt disabling/enabling bit in the PSW is not set; instead, it is set in the virtual interrupt controller, the software emulation of the interrupt controller. The detailed processing flow of the interrupt distribution routine is shown in Figure 2.

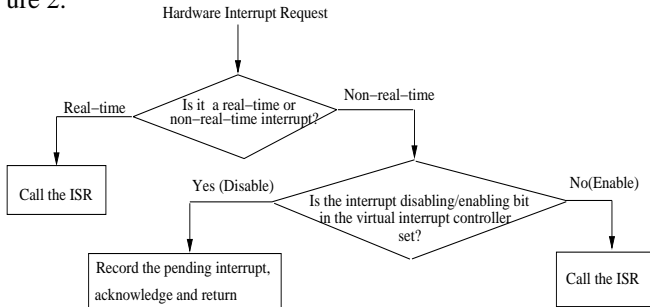


Figure 2. The interrupt processing flow of the interrupt distribution routine in RTAI.

As shown in Figure 2, in the interrupt distribution routine, when an interrupt request is responded, we first decide that it is a real-time interrupt or non-real-time interrupt. For a real-time interrupt, it will be directly served by calling its corresponding interrupt service routine. For a non-real-time interrupt, first we need to check whether or not the interrupt disabling/enabling bit in the virtual interrupt controller is set. If the bit is not set (the interrupts are not disabled), the interrupt will be served and the corresponding interrupt routine will be called; otherwise, the interrupt request is only recorded and acknowledged but not

really served. The interrupt requests recorded in the virtual interrupt controller will be served later when the interrupt disabling/enabling bit in the virtual interrupt controller is cleared.

This software emulation method causes some problems for RTAI. First, the interrupt disabling from the Linux task is processed by setting a flag in the interrupt-handling code based on the software emulation of interrupt controller without really disabling interrupts. Therefore, non-real-time interrupts of the Linux task can still be responded when they should be disabled during interrupt disabling. These unnecessary responses cause CPU overhead.

Moreover, the software method may cause some problems for systems with level-triggered interrupts. In a level-triggered hardware interrupt, the level (high or low) of the interrupt request line indicates whether or not there is an unserved interrupt. If a device wants to signal an interrupt, it drives the line to the active level, and then holds it at that level until the interrupt is serviced. Therefore, a normal processing procedure is to first serve and then acknowledge to drive the line to the inactive level. However, using the software emulation method, we have to first acknowledge an interrupt and then serve it when the interrupt disabling/enabling bit is set in the virtual interrupt handler. Therefore, it can not correctly handle level-triggered interrupts and may cause deal-lock or other problems for the system.

The software emulation method causes overhead for the real-time subsystem too. To provide predictable real-time services, real-time interrupt latency is one of the most important performance metrics for a hybrid system. In this paper, we define interrupt latency as the time interval from the point when an interrupt is generated by a device to the point when the corresponding interrupt service routine is reached. As predictability is the main concern in real-time systems, in the following, we will analyze the worse case execution time for real-time interrupt latency.

Without loss of generality, assume that there are N real-time interrupts, I_1, I_2, \dots, I_N . The priority order of N interrupts is $I_1 > I_2 > \dots > I_N$, which means that if I_1, I_2, \dots, I_N occur at the same time, the interrupt processing order is I_1, I_2, \dots, I_N . From the processing procedure shown in Figure 1, the real-time interrupt latency for interrupt I_K ($1 \leq K \leq N$) is related to the waiting time and the interrupt processing time. For the convenience of analysis, we further divide the the interrupt processing into two part: the distribution and the interrupt service parts. Based on this division, the real-time interrupt latency for interrupt I_K is shown in Figure 3.

As shown in Figure 3, the interrupt processing time of interrupt I_K , $T_P(K)$, consisting of two parts: the distribution time ($T_D(K)$) and the service time ($T_S(K)$), can be denoted as follows:

$$T_P(K) = T_D(K) + T_S(K).$$

$T_D(K)$ is the time interval from the point that the interrupt occurs to the point that the interrupt is served by reaching

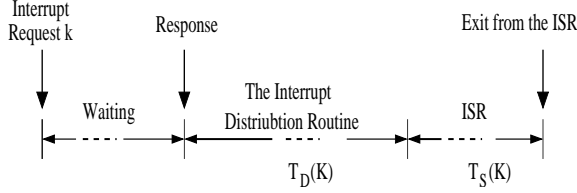


Figure 3. The real-time interrupt latency of interrupt I_K .

to its corresponding interrupt service routine. $T_S(K)$ is the interrupt service time in its corresponding interrupt service routine. Based on the definition, the worst-case real-time interrupt latency of interrupt I_K , $WCET(I_K)$, is the summation of the distribution time ($T_D(K)$) and the waiting time and is denoted as follows:

$$WCET(I_K) = T_D(K) + \text{The worst-case waiting time}$$

The worst-case waiting time can be divided into the following two parts:

1. The priority-related worst case: Considering the priority, the worst case is that interrupt I_K occurs at the same time as I_1, I_2, \dots, I_{K-1} , all interrupts that have higher priority than I_K , do. Based on the priority, I_K can only be processed after the processing for I_1, I_2, \dots, I_{K-1} has been finished. The worst case waiting time of this part, therefore, is $\sum_{i=1}^{K-1} T_P(i) = \sum_{i=1}^{K-1} \{T_D(i) + T_S(i)\}$
2. The interrupt-disable-related worst case: When I_1, I_2, \dots, I_K occur, the system may be in one of the following four states in which the hardware interrupts are disabled:
 - (1) The system just enters into the interrupt distribution routine. Therefore, we have to wait $\text{Max}\{T_D\}$ in the worst case until the interrupts are enabled.
 - (2) The system just enters into a real-time interrupt service routine. Therefore, we have to wait $\text{Max}\{T_S\}$ in the worst case until the interrupt service is finished.
 - (3) The system just enters into a critical section in a real-time task. Therefore, we have to wait $\text{Max}\{T_C\}$ in the worst case until the system exits from the critical section, where T_C represents one of the possible time intervals in which the hardware interrupts are disabled caused by entering a critical section in a real-time task.
 - (4) The system just finishes executing an operating system trap instruction in which all interrupts are automatically disabled. Therefore, we have to wait $\text{Max}\{T_T\}$ in the worst case until the interrupts are enabled, where T_T represents one of the possible time intervals from the point when the interrupt is disabled to the point when the interrupt is enabled caused by executing the trap instruction.

The worst case waiting time of this part, therefore, is $\max\{\max\{T_D\}, \max\{T_S\}, \max\{T_C\}, \max\{T_T\}\}$.

Thus, we obtain the worst case interrupt latency of interrupt I_K , $WCET(I_K)$, as follows:

$$WCET(I_K) = T_D(K) + \sum_{i=1}^{K-1} T_D(i) + \sum_{i=1}^{K-1} T_S(i) + \max\{\max\{T_D\}, \max\{T_S\}, \max\{T_C\}, \max\{T_T\}\} \quad (1)$$

From the above equation, the worst-case real-time interrupt latency is related to T_D , T_S , T_C and T_T , especially the first two. T_S , the processing time for an interrupt service routine, is decided by the interrupt processing and scheduling methods in the real-time subsystem. T_C , the waiting time for entering a critical section in a real-time task, is decided by the real-time subsystem. T_T , the delay caused by the trap instruction, is decided by the method to deal with system calls in the time-sharing subsystem. While it is closely related to the O.S. design for optimizing T_S , T_C and T_T , T_D , the delay caused by the interrupt distribution routine, can be optimized by changing the interrupt distribution method.

From the equation, we can see that T_D has the huge influence for the worst-case real-time interrupt delay. In the interrupt distribution routine of RTAI, we need to separate real-time/non-real-time interrupts and emulate the functions of the interrupt controller with the virtual interrupt controller. This makes it run slower and difficult to be locked in the cache with the big code size. As this distribution method is applied in most hybrid systems, it causes a general problem. Next, we will propose a scheme to solve it.

3 Implementing Hybrid Systems with Two-Level Hardware Interrupts

As shown above, several problems are caused by using the interrupt-handling code to separate real-time and non-real-time interrupts and emulate interrupt controller. To solve these problems, we propose to implement hybrid operating systems based on two-level hardware interrupts. In this section, we first discuss the methodology and key issues of implementing a hybrid system based on 2-level interrupts. Then we analyze the worst-case real-time interrupt latency based on this scheme.

Our two-level-interrupt hybrid-system implementation scheme is based on the two-level hardware interrupt architecture as shown in Figure 4. In Figure 4, real-time and non-real-time interrupts are separated by hardware and processed through different interrupt request entries. Therefore, in the interrupt distribution routine, we do not need to separate them, and as we can disable real-time/non-real-time interrupts independently, we do not need to emulate interrupt controller. A similar two-level hardware interrupt architecture has been provided in high-end embedded processors such as those based on the ARM architecture [26]. Next we will discuss the key issues to implement hybrid systems based on this two-level hardware interrupt architecture.

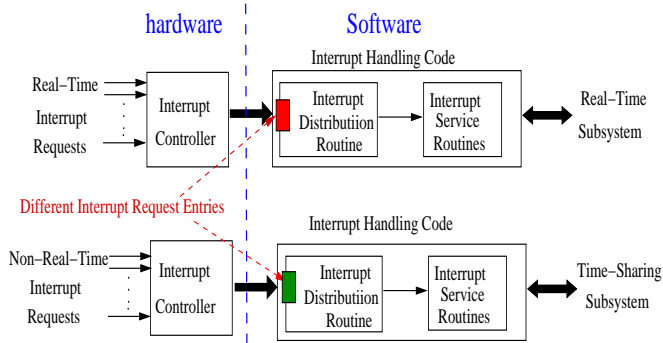


Figure 4. The two-level hardware interrupts.

With two-level hardware interrupts, it is relatively easy to implement hybrid systems. It is easier to implement interrupt distribution routines for both real-time and time-sharing subsystems. There is no need to separate real-time/non-real interrupts in the interrupt distribution routine since this has been done by hardware. The interrupts for the time-sharing subsystem can be disabled/enabled freely since it will not influence the interrupts for the real-time subsystem. Therefore, the problem caused by using software to emulate interrupt controller without really disabling interrupts can be solved. In the interrupt distribution routine, there is also no need to emulate interrupt controller. Thus, the interrupt distribution routines for both the real-time and time-sharing subsystems can be simplified and run faster. Correspondingly, we can reduce interrupt latency for both the real-time and time-sharing subsystems.

The predictability of the system can be improved with this scheme. First, the real-time interrupt distribution routine is simpler and can be implemented with smaller code size. Therefore, it is more possible for us to lock it into the cache. In this way, we can reduce its execution time and improve the predictability. Second, even when non-real-time interrupts are disabled, we can still preempt the time-sharing subsystem task or non-real-time interrupts through the real-time interrupt entry. Therefore, we can reduce the waiting time of a real-time interrupt.

Compared with the worst-case real-time interrupt latency from Equation 1, with the two-level hardware interrupts, T_T (the waiting time caused by the trap instruction) can be removed since we can preempt the Linux task through the real-time interrupt request entry. The real-time interrupt distribution delay can be reduced since the code is simplified by removing the real-time/non-real-time interrupt separation and software-emulation for interrupt controller. Therefore, with our scheme, we can obtain the equation of the worst-case real-time interrupt latency for interrupt I_K , $WCET(I_K)$, as follows:

$$\sum_{i=1}^K T'_D(i) + \sum_{i=1}^{K-1} T'_S(i) + \max\{\max\{T'_D\}, \max\{T'_S\}, \max\{T'_C\}\}$$

From the equation, we can see that there is no component related to the time-sharing (Linux) subsystem. T'_D is

smaller than T_D in Equation 1 as the real-time interrupt distribution routine in our scheme is simpler than the interrupt distribution routine in RTAI.

We do not need to use the software to emulate interrupt controller with this scheme. Therefore, we need to find different methods to implement some techniques that are implemented based on the software-emulation interrupt controller. For example, in RTAI, the communication between a real-time task and the Linux task is based on Real Time FIFO (First-In-First-Out) that is implemented based on the virtual interrupt controller. In Section 4, we discuss this problem and give a solution using spin locks.

4 System Implementation

Based on the two-level hardware interrupt scheme, we implement a hybrid system called RTLinux-THIN (Real-Time LINUX with Two-level Hardware INterrupts) on the ARM architecture [26] by combining ARM Linux kernel 2.6.9 and $\mu C/OS-II$ [16], two widely-used kernels with source code available in embedded and real-time fields. In this section, we first introduce the two-level interrupts in the ARM architecture. Then, we present the system structure and implementation details of our RTLinux-THIN system.

4.1 IRQ and FIQ in the ARM Architecture

In the ARM architecture, two-level hardware interrupts, IRQ (Interrupt Requests) and FIQ (Fast Interrupt Requests), are provided. All external interrupts are usually mapped to IRQ, which is the case in most operating systems. However, FIQ provides a faster method to serve interrupt requests. First, FIQ has higher priority than IRQ and other exceptions such as software interrupt exception, undefined instruction exception and data access abortion. Therefore, FIQ can preempt all other interrupts. Second, FIQ has its private registers (R8-R14), and less registers need to be protected in its interrupt distribution routine. In our implementation, non-real-time interrupts are mapped to IRQ and real-time interrupts are mapped to FIQ.

4.2 The System Structure of RTLinux-THIN

Based on IRQ and FIQ, we implement RTLinux-THIN by combining $\mu C/OS-II$ and ARM Linux. The system structure of the RTLinux-THIN is shown in Figure 5. The right-hand side of RTLinux-THIN is the real-time area that is managed by the $\mu C/OS-II$ kernel based on FIQ. The non-real-time area in the left-hand side is managed by the Linux kernel based on IRQ. In RTLinux-THIN, $\mu C/OS-II$ kernel is compiled as a part of the Linux kernel and works in the kernel mode. Moreover, $\mu C/OS-II$ is the scheduler for the whole system, and the Linux kernel is treated as the idle task with the lowest priority.

4.3 Implementation Details

Next, we introduce some key implementation issues including interrupt setting, real-time scheduling, and the communication between real-time tasks and non-real-time tasks.

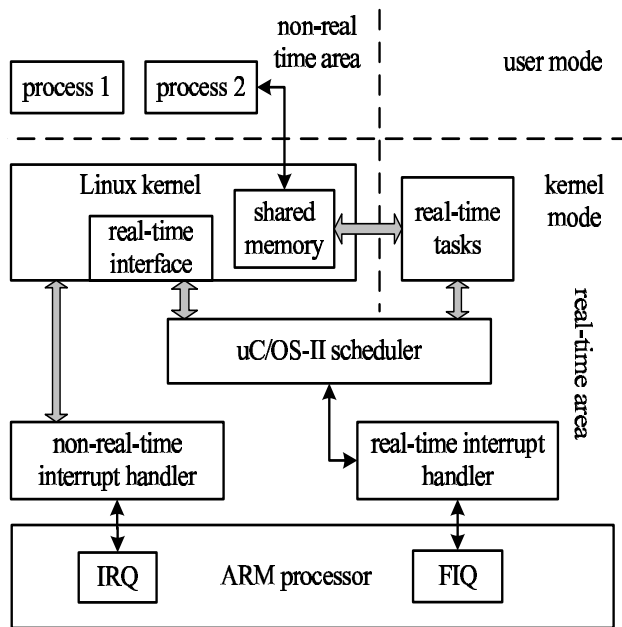


Figure 5. The system structure of RTLinux-THIN by combining μ C/OS-II and ARM Linux.

4.3.1 Interrupt Setting

According to the ARM architecture, all exception routine entries including IRQ, FIQ and software interrupts are put into a fixed virtual address by function *trap_init* at the Linux kernel startup. In the Linux, FIQ is not used, and it is disabled when the Linux kernel is booted. Moreover, there is no code involving with FIQ in the kernel. Therefore, the initialization procedure of IRQ for the Linux does not need to change. For FIQ setting, we need to set its environment correctly and then enable it in the Linux booting code.

In RTLinux-THIN, basically, we set the environment for FIQ as follows: (1) Set up the stack for FIQ. A private stack pointer is provided for FIQ and it must be set correctly. (2) Initialize the ISR descriptor table for FIQ so the corresponding interrupt service routines can be called correctly. This must be done at the startup and updated with real-time ISRs. (3) Enable FIQ.

FIQ is set and used only for the real-time kernel without any effect for the Linux. In the setting, we ensure that FIQ can be triggered and return to the original status regardless of the current mode of the Linux.

4.3.2 Real-Time Scheduling

As mentioned above, in RTLinux-THIN, the μ C/OS-II kernel is the scheduler for the whole system, and the Linux kernel is treated as the idle task with the lowest priority. The real-time scheduling is implemented based on the scheduler of μ C/OS-II. The priority order is set as follows: real-time interrupts > real-time tasks > non-real time interrupts > non-real-time tasks. Based on this priority order, a real-time task from the μ C/OS-II kernel has higher priority than non-real time interrupts. Therefore, the scheduling

will happen at the end of interrupt service routines of FIQ as the traditional μ C/OS-II does. When a real-time task is running, the interrupts from IRQ are disabled. In this way, we can guarantee the real-time performance of real-time tasks.

4.3.3 The Communication Between Real-Time and Non-Real-Time Tasks

In RTLinux-THIN, the Linux kernel as a task can communicate with real-time tasks by utilizing synchronization functions of μ C/OS-II such as *semaphore*, *mailbox*, *message queue*, *memory block*, etc. On the counterpart, the μ C/OS-II kernel and real-time tasks can easily access functions/resources of the Linux kernel as they are compiled in the same address space. However, as the μ C/OS-II kernel and real-time tasks have higher priority than the Linux kernel, it is possible that the Linux kernel is re-entered in some cases. Therefore, when utilizing the services provided by the Linux kernel, the μ C/OS-II kernel and real-time tasks need to consider that certain sections in the Linux kernel are exclusively executable. An example is given below to explain this.

Assume that there are two tasks, A (a real-time(μ C/OS-II) task) and B (the Linux kernel), to communicate with each other using a bidirectional FIFO (First In First Out) queue. Consider two scenarios as follows:

- *Scenario 1: Task A reads data from the FIFO, and task B writes data to it. If there is no data in the FIFO, task A needs to wait until data are produced by task B. In this scenario, we can use a *semaphore/mailbox* of μ C/OS-II for the synchronization. Basically, task A can be hung up and wait for the *semaphore* when there is no data in the FIFO, while task B can put data into the FIFO and post the *semaphore* to wake up task A. As B (the Linux kernel) is the task with the lowest priority, when it is scheduled to be executed, we can guarantee that there is no any ready real-time task.*
- *Scenario 2: Task B reads data from the FIFO, and task A writes data to it. If there is no data in the FIFO, when task B read from it, it should be hung up and cause the scheduler of the Linux kernel rather than μ C/OS-II to do scheduling. So *semaphore* or *mailbox* of μ C/OS-II can not be used here, and we can only use the communication mechanism such as *waiting queue* provided by the Linux kernel. Then when task A puts data into the FIFO, it has to use the waking-up functions provided the Linux kernel to activate task B. However, as task A has higher priority, when A was scheduled, it may preempt the Linux task when the Linux kernel was using the same waking-up function. As the wake-up function is non-reentrant in the Linux kernel, task A should wait until the Linux kernel exits from it.*

From this example, we can see that how to solve the exclusive execution of sections in the Linux kernel is the key to implement the communication between real-time and

non-real-time tasks. In RTAI, this is solved based on the virtual interrupt controller. With our two-level hardware interrupts, we do not need to use software-emulation interrupt controller. In RTLinux-THIN, we solve this problem using spin locks.

Spin locks are designed for Linux kernel to protect sections with exclusive execution in SMP (Symmetrical Multi-Processing). They are also used in preemptive Linux kernels such as Linux 2.6 kernel to mark non-preemptive regions. Therefore, we can use *spin locks* to protect sections with exclusive execution in the Linux kernel. We implement this by adding a *semaphore* of $\mu\text{C}/\text{OS-II}$ into the *spin lock* structure of the Linux kernel with the initial value of one. In this way, when we enter an exclusively-executable section, the corresponding spin lock is held until we exit. Therefore, these sections can be protected. Based on this, we have implemented Real-Time FIFO in RTLinux-THIN.

5 Experiments

We implement our RTLinux-THIN on a hardware platform based on Intel PXA270 processor [12]. We conduct experiments with three real application programs: mplayer [20], Bonnie [4] and iperf [13], and compare statistical interrupt latency distributions in four system conditions: system idle, intensive memory accesses (decoding a fragment of MPEG4 video in mplayer), intensive IDE Disk accesses (evaluating the speed of the filesystem with a set of file operating benchmarks in Bonnie) and intensive network communication (measuring TCP/UDP bandwidth performance in iperf). In this section, we present and analyze the experimental results. We first introduce the experimental environment including hardware platform and operating systems we evaluate. Then we present the performance metric and our measurement method. Finally, the experimental results are given and discussed.

5.1 Experimental Environment

5.1.1 Hardware platform

The hardware platform we use is based on Intel PXA270 processor [12] which is a XScale-based core complying with the ARM architecture v5TE instruction set. In the platform, the processor is running at 520MHz with 64MB SDRAM, and the cache and eight-entry write buffer are enabled. The on-chip LCD controller is configured for VGA resolution with 16 bit color whose frame refresh frequency is about 70Hz.

5.1.2 Operating Systems Evaluated

We implement RTLinux-THIN and port ARM Linux 2.6.9 and RTAI on this hardware platform. As shown in Sections 2 and 3, interrupt distribution routines in a hybrid system have huge influence for interrupt latency. In Table 1, we compare the main functions and code sizes of the real-time interrupt distribution routines in RTLinux-THIN and RTAI based on this platform. in Table 1.

From Table 1, we can see that in order to lock the real-time interrupt routines into the cache on Intel PXA270

RTLinux-THIN

Functions	Number of Instructions	Number of Cache Lines
FIQ entry	1	1
__do_FIQ	13	5
asm_do_FIQ	23	
total	37	6

RTAI

Functions	Number of Instructions	Number of Cache Lines
IRQ entry	1	1
vector_IRQ	12	2
__irq_svc	32	4
__irq_usr	48	6
__adeos_handle_irq	151	33
__adeos_sync_stage	112	
total	356	46

Table 1. The real-time interrupt distribution routines of RTLinux-THIN and RTAI.

processor, 6 and 33 cache lines are needed for RTLinux-THIN and RTAI, respectively. We implement a version of RTLinux-THIN by locking its real-time interrupt distribution routine into the cache as it only takes 6 cache lines. The version with cache locking is called RTLinux-THINLock. Totally we conduct experiments on four systems: RTLinux-THINLock, RTLinux-THIN, RTAI and Linux.

5.2 Performance Metric and Measurement Method

Interrupt latency is the performance metric we focus on for evaluating the four systems. To get a complete performance evaluation, the statistical distribution of interrupt latency in a time period is used in our experiments, which means we will record each interrupt latency and calculate the statistical distribution within a time period. In the following, we show how to set up a hardware timer so hardware interrupts can be generated periodically and how to measure its interrupt latencies.

We will use a timer whose frequency is set up as 3.25MHz (clock cycles = 308 ns) to measure interrupt latency. Two registers in this timer, OSCAR (OS Timer Count Register) and OSMR (OS Timer Match Register), are used for calculating the time interval from the point when an interrupt occurs to the point when its corresponding ISR is reached. OSCAR is incremented in each clock cycle. When its value reaches to the value we preset in OSMR, an interrupt will be generated. In the ISR of this interrupt, we will read the current value of OSCAR into a register R. Then the interrupt latency can be calculated by $(R - \text{OSMR}) / 3.25 \text{ us}$. After we obtain this latency, we record it into an array in memory, and then we set up the value of OSMR with $\text{OSCAR} + 3076$ ($3076 = 10\text{ms} / 3.25\text{MHz}$) so as to gener-

ate next interrupt in about 10 ms. The accuracy of this measurement method is up to several tens of nanoseconds on the PXA270 processor (520 MHz), and the only derivation is caused by the time of executing two instructions in ISR (get the address of the OSCR, and load the value of OSCR into a register).

This interrupt is registered as a real-time interrupt by using `re_request_irq` in RTAI and `request_fiq` in our RTLinux-THIN and RTLinux-THINLock. We implement a device driver for this timer to provides a user interface. Based on this, in a user program, we can star/stop the interrupt-latency recording in the ISR of the timer, and read experimental results to the user space.

Interrupt latencies are obtained in four system conditions with three real application programs: `mplayer` [20], `Bonnie` [4] and `iperf` [13]. The four system conditions are: system idle, intensive memory accesses by decoding a fragment of MPEG-4 video in `mplayer`, intensive IDE Disk accesses by evaluating the speed of the filesystem with a set of file operating benchmarks in `Bonnie`, and intensive network communication by measuring TCP/UDP bandwidth performance in `iperf`. For each operating system, in each system condition, a user testing program is implemented to run for 10 minutes to obtain about 60,000 interrupt-latency samples. Each experiment is repeated 12 times, and then all results are used to generate the final statistical distribution.

5.3 Results and Discussions

Based on the above method, the statistical distributions of real-time interrupt latencies for four systems, RTLinux-THIN, RTLinux-THINLock, RTAI and Linux, are shown in Figures 6-9 for the four conditions, respectively. In each of the above figures, there is a two-dimensional plot where the X-axis denotes the time (unit: μs) with logarithmic scale and the Y-axis denotes the percentage. On each plot, there are four curves to denote the statistical distributions of real-time interrupt latencies for four systems ("THIN" for RTLinux-THIN, "THINLock" for RTLinux-THINLock, "RTAI" for RTAI, "Linux" for Linux), respectively.

For each point (x,y) on a curve, y denotes the percentage of the number of interrupt latencies that are less than or equal to x to the total number of interrupt latencies. For example, on the curve for RTLinux-THINLock in Figure 6, the point (4.34,100) denotes that the percentage of the number of the interrupt latencies that are less than or equal to 4.34 μs to the total number of the interrupt latencies is 100%. For each curve, the minimum and maximum interrupt latencies are denoted in the figure. Next, we will present and discuss four plots for four different conditions, respectively.

5.3.1 System Idle

Four curves in Figure 6 show the statistical interrupt latency distributions for the four operating systems, respec-

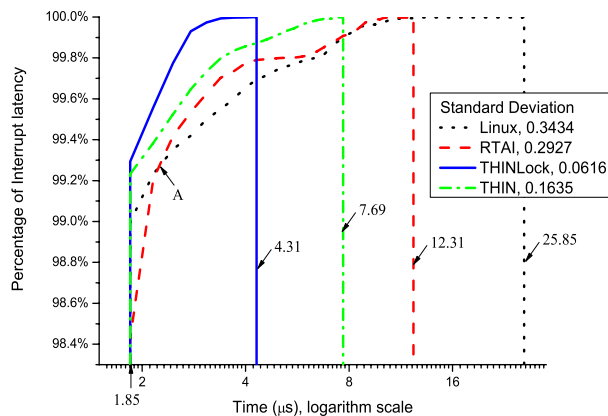


Figure 6. The interrupt latency distributions for RTLinux-THIN, RTLinux-THINLock, RTAI and Linux when the system is idle.

tively. When the system is idle, the system performance is very predictable, and cache misses and interrupt conflicts hardly happen. Therefore, we can see that over 98% of interrupt latencies reaches to the minimum interrupt latency. For the worst case, 7.69 μs and 4.31 μs are obtained by RTLinux-THIN and RTLinux-THINLock, respectively, while 12.31 μs by RTAI. Therefore, our RTLinux-THIN and RTLinux-THINLock provide better results than RTAI. RTLinux-THINLock provides the best result with its cache locking scheme.

5.3.2 Intensive Memory Accesses in mplayer

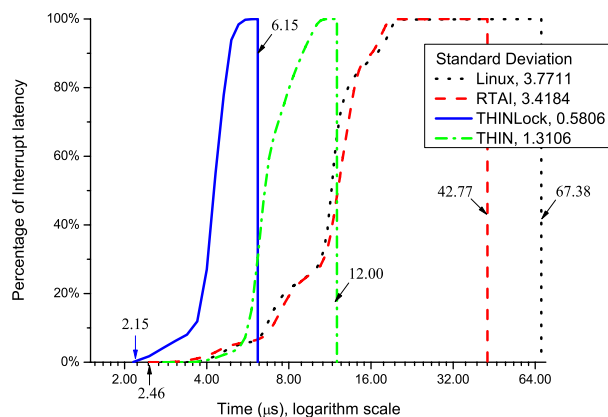


Figure 7. The interrupt latency distributions for RTLinux-THIN, RTLinux-THINLock, RTAI and Linux when a fragment of MPEG-4 video is decoded in mplayer [20].

Figure 7 shows the interrupt distributions when a fragment MPEG-4 video is decoded in RAM by *mplayer* [20]. Because the video is stored in RAM, interrupt latencies are mainly influenced by two factors, cache misses and system calls. These factors cause some influences for RTAI, and its worst-case real-time interrupt latency is 42.77 μs . With our two-level interrupt scheme, RTLinux-THIN shows better performance with 12.00 μs by completely avoiding the influence caused by system calls from Linux kernel. RTLinux-THINLock gives the best with 6.15 μs .

5.3.3 Intensive IDE Disk Accesses in Bonnie

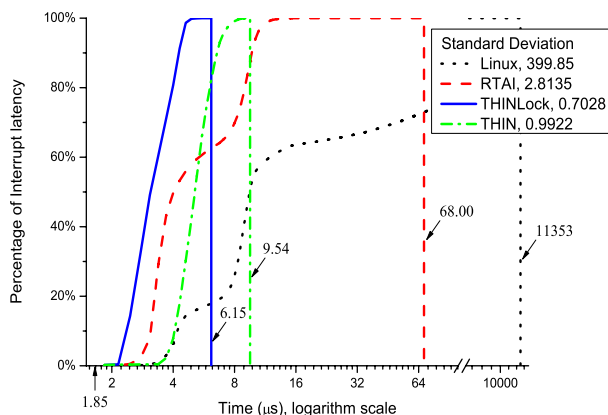


Figure 8. The interrupt latency distributions for RTLinux-THIN, RTLinux-THINLock, RTAI and Linux when the speed of the filesystem is evaluated with a set of file operating benchmarks in Bonnie [4].

Figure 8 shows the interrupt latency distributions for RTLinux-THIN, RTLinux-THINLock, RTAI and Linux when the speed of the filesystem is evaluated with a set of file operating benchmarks in Bonnie. Since the device driver for IDE disk in our platform does not use DMA, the interrupts may be disabled for very long time in the driver. Therefore, it posts a huge impact for the worst-case interrupt latency for Linux which is 11353 μs . In RTAI, this problem is solved using virtual interrupt controller; therefore, it shows a huge improvement over Linux with 68 μs . Our RTAI-THIN can RTAI-THINLock can provide further improvements with 9.54 μs and 6.15 μs , respectively.

5.3.4 Intensive Network Communication in iperf

Figure 9 shows the interrupt latency distributions for RTLinux-THIN, RTLinux-THINLock, RTAI and Linux when the TCP/UDP bandwidth performance is measured by iperf. When iperf is running, the interrupt handling code is frequently invoked. Therefore, the worst-case real-time interrupt latency from RTAI is 23.08 μs that is mainly

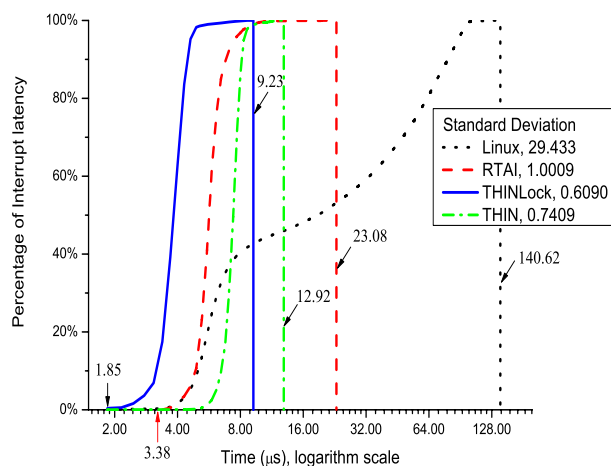


Figure 9. The interrupt latency distributions for RTLinux-THIN, RTLinux-THINLock, RTAI and Linux when the TCP/UDP bandwidth performance is measured by iperf [13].

caused by its interrupt distribution routine. Our RTLinux-THIN improves this to 12.92 μs since its distribution routine is simpler and runs faster. RTLinux-THINLock further improves this to 9.23 μs by reducing cache misses with cache locking.

From Figures 6-9, we can see that RTLinux-THIN and RTLinux-THINLock improve the worst-case interrupt latencies in different situations compared with RTAI. The worst-case real-time interrupt latencies from our RTLinux-THIN and RTLinux-THINLock do not vary a lot with different situations. The ranges are 4.31-9.23 μs for RTLinux-THINLock and 7.69-12.92 μs for RTLinux-THIN, while it is 12.31-68 μs for RTAI. So our RTLinux-THIN provides better predictability.

6 Conclusion

In this paper, we proposed to implement hybrid systems with two-level hardware interrupts. We first investigated the interrupt processing in hybrid systems and analyzed the worst-case interrupt latency of RTAI. Then we discussed the key issues for implementing a hybrid system based on two-level hardware interrupts. We implemented a hybrid system called RTLinux-THIN (Real-Time LINUX with Two-level Hardware INterrupts) on the ARM architecture by combining ARM Linux kernel 2.6.9 and $\mu\text{C}/\text{OS-II}$ based on this scheme.

We implemented RTLinux-THIN on a hardware platform based on Intel PXA270 processor [12]. We conducted experiments with three real application programs: *mplayer* [20], *Bonnie* [4] and *iperf* [13] and compared statistical interrupt latency distributions in four system conditions: idle, intensive memory access, intensive IDE Disk access and intensive network communication for RTLinux (with

or without cache locking), RTAI and Linux. The experimental results show that RTLinux-THIN improves real-time interrupt latencies and provides better predictability.

References

- [1] A. Arnaud and I. Puaut. Towards a predictable and high performance use of instruction caches in hard real-time systems. In *Proc. of the work-in-progress session of the 15th Euromicro Conference on Real-Time Systems*, pages 61–64, Porto, Portugal, July 2003.
- [2] A. Arnaud and I. Puaut. Dynamic instruction cache locking in hard real-time systems. In *Proc. of the 14th International Conference on Real-Time and Network Systems (RNTS)*, Poitiers, France, May 2006.
- [3] K. W. Batchner and R. A. Walker. Interrupt triggered software prefetching for embedded cpu instruction cache. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 91–102, Washington, DC, USA, 2006. IEEE Computer Society.
- [4] Bonnie. <http://www.garloff.de/kurt/linux/bonnie/>.
- [5] A. Campoy, A. Perles, F. Rodriguez, and J. Busquets-Mataix. Static use of locking caches vs. dynamic use of locking caches for real-time systems. *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on*, 2:1283–1286, 2003.
- [6] M. Campoy, A. Ivars, and J. Busquets-Mataix. Static use of locking caches in multitask preemptive real-time systems. In *Proceedings of IEEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, 2001.
- [7] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 257–270, Washington, DC, USA, 2006. IEEE Computer Society.
- [8] Z. Deng, J. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *In Proceedings of the Ninth Euromicro Workshop on Real-Time System*, pages 191–199, Jun. 1997.
- [9] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, page 308, Washington, DC, USA, 1997. IEEE Computer Society.
- [10] X. A. Feng and A. K. Mok. A model of hierarchical real-time virtual resources. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 26, Washington, DC, USA, 2002. IEEE Computer Society.
- [11] T. A. Henzinger and C. M. Kirsch. The embedded machine: Predictable, portable real-time code. In *Proceedings of the International Conference on Programming Language Design and Implementation*, pages 315–326. ACM Press, 2002.
- [12] Intel, Inc. *Intel PXA27x Processor Family Developer's Manual*, Jan 2006.
- [13] iperf. <http://dast.nlanr.net/projects/iperf>.
- [14] C. Kirsch, M. Sanvido, and T. Henzinger. A programmable microkernel for real-time systems. In *Proc. ACM/USENIX Conference on Virtual Execution Environments (VEE)*. ACM Press, 2005.
- [15] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 256, Washington, DC, USA, 1999. IEEE Computer Society.
- [16] J. J. Labrosse. *MicroC/OS-II: The Real-Time Kernel, 2nd Edition*. CMP BOOKS, Apr. 2002.
- [17] I. Lee and I. Shin. Periodic resource model for compositional real-time guarantees. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, 2003.
- [18] P. Mantegazza, E. L. Dozio, and S. Papacharalambous. Rtai: Real time application interface. *Linux Journal*, 2000(72es):10, 2000.
- [19] A. K. Mok, X. A. Feng, and D. Chen. Resource partition for real-time systems. In *RTAS '01: Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*, page 75, Washington, DC, USA, 2001. IEEE Computer Society.
- [20] mplayer. <http://www.mplayerhq.hu/>.
- [21] S. Oikawa and R. Rajkumar. Linux/rk: A portable resource kernel in linux. In *IEEE Real-Time Systems Symposium Work-In-Progress*, 1998.
- [22] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 114, Washington, DC, USA, 2002. IEEE Computer Society.
- [23] QLinux. <http://www.cs.umass.edu/lasp/software/qlinux/>.
- [24] RED-Linux. <http://linux.ece.uci.edu/rea-linux/>.
- [25] J. Regehr and J. A. Stankovic. Hls: A framework for composing soft real-time schedulers. In *IEEE Real-Time Systems Symposium*, pages 3–14. IEEE Computer Society, 2001.
- [26] D. Seal. *ARM Architecture Reference Manual, 2nd Edition*. Addison-Wesley, Nov. 2000.
- [27] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *RTAS '98: Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, page 112, Washington, DC, USA, 1998. IEEE Computer Society.
- [28] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 154, Washington, DC, USA, 2003. IEEE Computer Society.
- [29] K. Yaghmour. Adaptive domain environment for operating systems. Jun. 2002.
- [30] V. Yodaiken and M. Barabanov. Real-time linux. In *Linux Applications Development and Deployment Conference(USELINUX)*, Jan. 1997.
- [31] Y. Zhang and R. West. Process-aware interrupt scheduling and accounting. *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 191–201, 2006.