The Limitations of Fixed-Priority Interrupt Handling in PREEMPT_RT and Alternative Approaches *

Glenn A. Elliott and **James H. Anderson**
Department of Computer Science, University of North Carolina at Chapel Hill
201 S Columbia St., UNC-CH, Chapel Hill, NC, 27599-3175
{gelliott, anderson}@cs.unc.edu

Abstract

Threaded interrupt handling is a common technique used in real-time operating systems since it increases system responsiveness and reduces priority inversions. The PREEMPT_RT Linux kernel patch introduces aggressive threaded interrupt handling into the Linux kernel. However, under PREEMPT_RT, interrupt handling threads must be assigned a single fixed scheduling priority. This can become a significant limitation when an interrupt-generating device is shared by threads of differing priorities. In this paper, we show that there is no good option for assigning a single fixed priority to an interrupt handling thread in such cases. We then survey alternative approaches from academic literature and commercial real-time operating systems to inspire new solutions in PREEMPT_RT.

1 Introduction

An interrupt is a hardware signal issued from a system device to a system CPU. The name "interrupt" is apt because the receiving CPU must immediately suspend (interrupt) its normal thread of execution to instead execute a segment of code responsible for taking the appropriate actions to process the interrupt. An interrupted thread may only resume execution after the interrupt handler has completed.

Interrupts require careful implementation and analysis in real-time systems. In uniprocessor and partitioned multiprocessor systems, an interrupt handler can be modeled as the highest-priority real-time thread [4, 6], though the unpredictable nature of interrupts in some applications may require conservative analysis. Such approaches can be extended to multiprocessor systems where threads may migrate between CPUs [1]. However, in such systems, the subtle difference between an interruption and preemption creates an additional concern: an interrupted thread cannot immediately migrate to another CPU since the interrupt handler temporarily uses the interrupted thread's program stack. As a result, conservative analysis must also be used when accounting for interrupts in these systems too. A real-time system, both in analysis and in practice, benefits greatly by minimizing interruption durations. Split interrupt handling is a common way of achieving this, even in non-real-time systems.

Under general *split interrupt handling*, interrupt processing is split into two parts: a *top-half* and a *bottom-half*. The top-half is executed immediately when an interrupt is received, interrupting the normally scheduled thread as described earlier. The top-half performs the minimum amount of processing necessary to ensure proper functioning of hardware; additional work to be carried out in response to an interrupt is deferred to the bottom-half. In most real-time systems, bottom-halves are processed by dedicated threads of execution that are scheduled with an appropriate priority. Split interrupt handling minimizes the duration of interruption and deferred work competes fairly with other threads for CPU time. However, determining an "appropriate" scheduling priority for the deferred work is non-trivial. The correct choice depends upon how the interrupt-raising device is used and the priority of threads using that device.

The prioritization of deferred work from interrupt handlers in the real-time Linux kernel patch, PREEMPT_RT, is straightforward: individual threads dedicated to processing deferred work from interrupt handlers are assigned a single fixed priority. Unfortunately, there are limitations to this simple approach if an interrupt-generating device is shared by client threads of differing priorities. If an interrupt thread is given too low a priority, then high-priority device-using threads may be delayed from execution while waiting for their associated

*Work supported by NSF grants CNS 1016954 and CNS 1115284; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

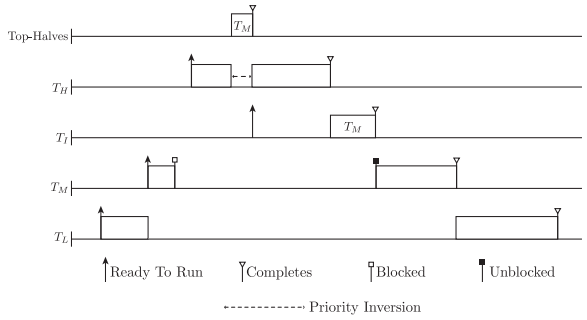

FIGURE 1: *Fixed-priority assignment when an I/O device is used by a single thread.*

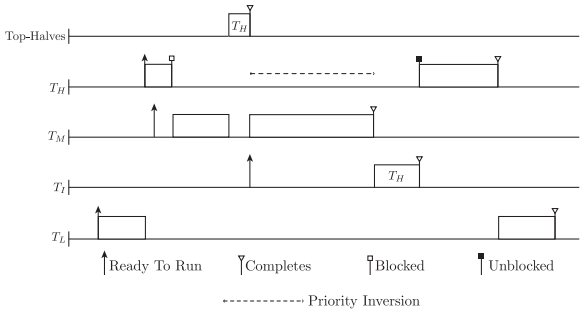

FIGURE 2: *T_H may suffer an unbounded priority inversion if $prio(T_I)$ is too low.*

interrupts to be processed. If an interrupt thread is given too high a priority, then the processing of interrupts associated with low-priority device-using threads delay the execution of higher-priority threads, even those that may not use a device.

We further explore these limitations and discuss alternative approaches in this paper. In Sec. 2, we illustrate the limitations of fixed-priority interrupt handling in PREEMPT_RT and demonstrate that they have a real impact on system performance by presenting a study of graphics processing units (GPUs) used to perform general purpose computation, an increasingly common practice. In Sec. 3, we briefly survey approaches to real-time interrupt handling that appear in academic literature and several commercial real-time operating systems. We conclude in Sec. 4.

2 Limitations of Fixed-Priority Handlers

PREEMPT_RT is a fixed-priority real-time operating system (RTOS) based upon POSIX standards. Under fixed-priority scheduling, each thread of execution is assigned a single fixed priority. In general, the system schedules the highest-priority threads that are ready to run. As discussed in Sec. 1, PREEMPT_RT uses threaded split interrupt handling. Though the organization of interrupt handling threads changes with the continued development of PREEMPT_RT, one fundamental design feature that has remained unchanged is that threaded interrupt handlers are also assigned a single fixed priority.

There are scenarios where this priority assignment method to interrupt handling threads is sufficient. Consider a uniprocessor real-time system with three independent threads, T_H, T_M, and T_L. T_H is assigned a high priority of five ($prio(T_H) = 5$), T_M is assigned a middle priority of three ($prio(T_M) = 3$), and T_L is assigned a low priority of one ($prio(T_L) = 1$). Suppose T_M issues a command to an I/O device and suspends from execution until an interrupt from the device, indicating completion of the operation, has been processed. T_M cannot resume execution until the bottom-half of the interrupt has completed. What priority should be assigned to the interrupt handling thread, denoted T_I, that will do this work? In order to avoid causing interface with other threads, T_I may have any priority greater than or equal to T_M but less than the priority of T_H. Specifically, the priority of T_I may be three or four. (To avoid ambiguity in scheduling, interrupt handling threads are commonly given a priority slightly greater than their dependent threads. [7]) As depicted in Fig. 1, with this priority assignment, the operations of T_M have less impact on T_H (priority inversions due to top-halves are unavoidable); T_M and T_I only receive processing time when T_H is not ready to run. Likewise, T_L can in no way delay the execution of T_I. Because of the lack of interference, this system is also easy to analyze. However, the situation changes when the I/O device is shared by different threads of differing priorities.

Let us reconsider the prior scenario with one change: suppose T_H and T_L share the I/O device simultaneously, and T_M does not use the device at all. Does this affect the priority assigned to T_I? Indeed it does. If the priority of T_I is less than T_M, then T_H can experience (potentially unbounded) priority inversions. For example, this may occur when T_H suspends from execution after issuing a command to the I/O device and suspends to wait for completion of the command. The interrupt indicating that the operation has completed may be received, top-half executed, and bottom-half deferred to T_I, but if $prio(T_I) < prio(T_M)$ and T_M is scheduled, then T_I cannot execute and unblock T_H until T_M gives up the processor. Thus, T_H indirectly suffers a priority inversion with respect to T_M. Such a scenario is illustrated in Fig. 2. *Observe that the duration of this inversion is largely not dependent upon the time it takes to execute the interrupt bottom-half*, but rather upon the potentially unbounded duration between when T_I is ready to run and T_M relinquishes the processor. This dependency can break analysis and real-time predictability may not be ensured.

The potential for unbounded priority inversion forces an alternative priority assignment where T_I is assigned a priority great enough to ensure T_H cannot suffer this par-

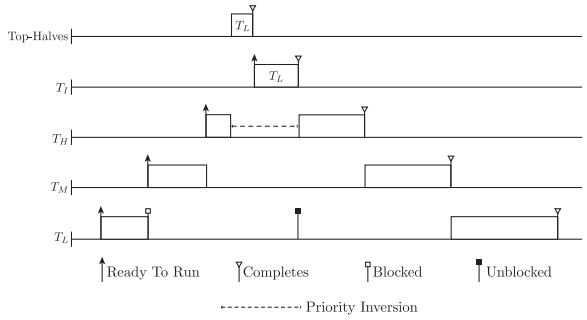

FIGURE 3: *T_H may suffer a bounded priority inversion when T_I processes a bottom-half for T_L.*

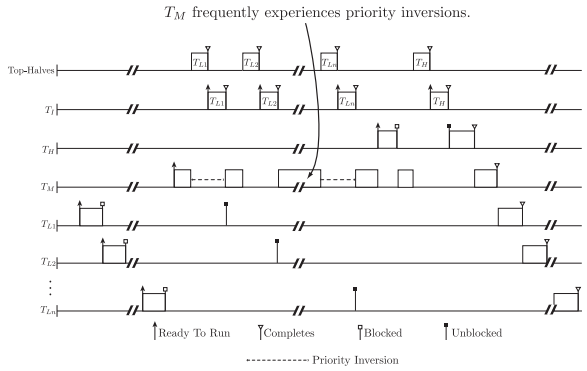

FIGURE 4: *A pathological scenario for fixed-priority interrupt handling.*

ticular priority inversion. In general, the priority of T_I must be no less than the highest priority thread that may depend upon T_I. In our scenario, $prio(T_I) = 6$ would suffice. However, this assignment introduces a different priority inversion scenario. What happens when T_I processes a bottom-half for which T_L blocks, as depicted in Fig. 3? Since T_I has the greatest priority, it is immediately scheduled whenever it is ready to run, so the bottom-half for T_L is immediately processed, resulting in the preemption of any other threads, including T_H. This is another priority inversion from which any threads with priorities less than T_I, but greater than T_L, may suffer. The only advantage to using a greater priority for T_I is that at least these inversions are of bounded duration—one inversion only lasts as long as the execution time of one bottom-half. However, the priority assignment that we have been forced to use is susceptible to pathological cases. Suppose that T_H rarely uses the I/O device and T_L uses it very frequently, generating many interrupts. Or, suppose there are many low-priority threads that use the I/O device. In either case, T_H and T_M may experience many priority inversions, as illustrated in Fig. 4.

Empirical Study. The limitations to fixed-priority interrupt handling can be shown to have an impact in practice. In prior work [3], with results partially replicated here, we studied fixed-priority interrupt handling in PREEMPT_RT and its relation to general purpose computation carried out on graphics processing units, a practice called GPGPU.

GPGPU technology allows general C-like program code for data parallel problems to be executed very efficiently, both in terms of speed and power, on graphics hardware. However, unlike CPUs, GPUs are not independently schedulable processors since they are interfaced to the host system as an I/O device, even in on-chip architectures. GPGPU has found applications in a wide array of domains, including real-time systems. One such application is in future automotive systems, where GPUs may be used to process data-intensive sensor feeds and perform compute-intensive computer vision algorithms [2].[1]

In order to study the effects of fixed-priority interrupt handling on real-time GPU workloads, we executed a workload of CPU-only and GPU-using tasks on a dual-socket six-cores-per-socket Intel Xeon X5060 CPU system running at 2.67GHz that is equipped with eight NVIDIA GTX-470 GPUs. The platform has a NUMA architecture of two NUMA nodes, each with six CPU cores and four GPUs apiece.

The workload was scheduled using the clustered rate-monotonic (RM) algorithm since it is easily supported by fixed-priority schedulers. Counting semaphores were used to protect access to GPUs in order to prevent the closed source driver from using its own (non-real-time) resource-arbitration algorithms. The workload consisted of 50 periodic tasks. A periodic task is made up of a sequence of jobs that arrive at regular intervals. In this experiment, each job had a deadline equal to this interval, such that the job was to be completed before the arrival of the next job of the task. Each task was scheduled as a single thread. Of the 50 tasks, two were GPU-using tasks that consume $2ms$ of CPU time and $1ms$ of GPU time with a period of $19.9ms$; 40 were CPU-only tasks that consume $5ms$ of CPU time with a period of $20ms$; and finally, eight were additional GPU-using tasks that consume $2ms$ of CPU time and $1ms$ of GPU time with a period of $20.1ms$. The tasks were evenly partitioned between the system's NUMA nodes. Unique priorities were assigned to each task according to task period.

[1] Interestingly, PREEMPT_RT is likely the only mainstream RTOS in position to support these applications since high-performance GPU drivers are currently limited to Windows, Mac, and Linux-based platforms. Unfortunately, high-performance drivers are currently only developed by GPU manufactures and are closed source. These developers have yet to completely embrace PREEMPT_RT. For example, current drivers from NVIDIA can only be installed without modification in versions of PREEMPT_RT that are several years old. Minor modifications to the GPL glue-layer distributed with the closed source driver are necessary in order to install these drivers in newer versions of PREEMPT_RT. Of course, this is an unsupported configuration. Nevertheless, PREEMPT_RT likely offers the best real-time performance among possible platforms. It is also especially attractive since it can support Android "infotainment" applications [9].

	Low Prio. Interrupts	High Prio. Interrupts
Average % of Job Deadline Misses Per Task		
CPU-Only Tasks	12.5%	12.5%
GPU-Using Tasks	10.1%	8.5%
Average Response Time as % of Period		
CPU-Only Tasks	22,474.5%	24,061.0%
GPU-Using Tasks	23,066.1%	34,263.5%

TABLE 1: *Average number of deadline misses per task and average job response times (expressed as a percentage of period).*

This workload and prioritization represents the pathological case discussed earlier. Here, the highest and lowest priority tasks share GPUs and the interrupt handling tasks, which each have a single fixed priority. Unrelated CPU-only tasks are sandwiched between these GPU-using tasks. If all tasks had equal priority, then under RM scheduling, priorities could be reassigned such that CPU-only tasks have priorities strictly greater or strictly less than those of GPU-using tasks. However, though task periods are close to being equal, it is not the case here.

The workload was executed under several system configurations, including (1) PREEMPT_RT, with GPU-interrupt priorities set below that of any other real-time task; and (2) PREEMPT_RT with GPU-interrupt priorities greater than the greatest GPU-using task. PREEMPT_RT was based upon the 2.6.33 Linux kernel, real-time patch `rt30`, which was the most recent kernel supported by PREEMPT_RT at the time of our evaluation. This workload was executed 25 times for each system configuration for a duration of 60 seconds each. Measurements were recorded consistently on each platform.

Table 1 gives the average percentage of deadlines missed, as well as average response times (as percent of period), for CPU-only and GPU-using tasks under the PREEMPT_RT scenarios. The percentage of deadlines missed is useful for comparing schedulability. Response time measurements express the timeliness of job completions (or severity of a deadline misses).

A deadline miss occurs if a job does not complete within one period of its release time. We avoid penalizing the response time of a subsequent job following a missed deadline by *shifting* the job's release point to coincide with the tardy completion time of the prior job. However, since these tests execute for a constant duration, frequently tardy tasks may not execute all their jobs within the allotted time; any jobs that have not completed (even those not yet released) by the end of a test are considered to have missed deadlines, though these jobs are not included in response time measurements.

As it can be observed in Table 1, there are no good

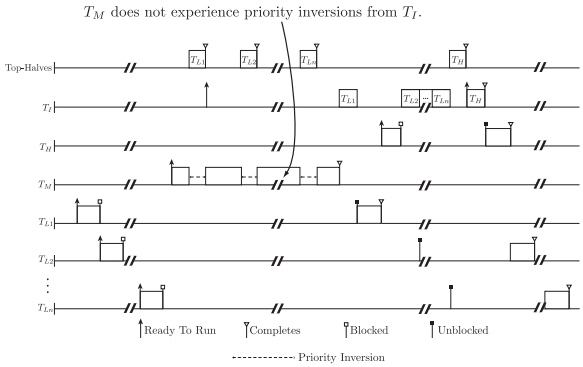

FIGURE 5: *T_I with a dynamic priority. T_I is only scheduled with a high priority when the device is in use by T_H.*

options for selecting a ***fixed*** priority for interrupt tasks shared by tasks of differing priorities. The high priority interrupt handlers (second column) causes all bottom-half task execution to preempt CPU-only work, directly increasing their response times with respect low priority interrupt handlers (first column). With high priority interrupt handlers, GPU interrupt execution is often on behalf of lower-priority GPU-using work, thus causing CPU-only work to experience priority inversions. Priority inversions also occur if interrupt priority is too low, resulting in the starvation of GPU-using work—deadline misses were more common for GPU-using tasks when low-priority interrupt handlers were used.

The Fundamental Issue and Desired Behavior. Fixed-priority interrupt handling clearly has limitations, but what is the fundamental issue causing this limitation? The problem is that interrupt threads can be scheduled with the wrong priority because interrupt threads are scheduled with a priorities not dependent upon the actual priority of pending work being processed. An interrupt for a low priority thread may be scheduled at too high a priority, or an interrupt for a high priority thread is scheduled at too low a priority. Both cases result in disruptive priority inversions. Instead, *the priority of the interrupt thread should be dynamic.*

Interrupt threads should be scheduled with a priority at least as great as the highest-priority thread *pending* on the interrupt-generating device—either waiting for access to the device, or waiting for an interrupt from the device to be processed. Let us return to our example introduced at the beginning of this section with threads T_H, T_M, and T_L. When T_H and T_L share an I/O device, the interrupt thread T_I is scheduled only with T_H's priority when the timely execution of T_H is dependent upon the timely execution of T_I. This is illustrated in Fig. 5.

We wish to point out that our definition for proper interrupt thread prioritization needs further refinement on globally scheduled systems (those where threads may migrate between CPUs); we explore this further in the

next section. We also realize that our definition does not cover cases when I/O devices deliver externally triggered messages, such as the network device in a real-time network server. Literature suggests that it is best to use a bandwidth-server to limit the total utilization these interrupts can put on the system [5]. In fact, this has already been explored using the SCHED_DEADLINE patch [8]. We will not consider this use-case further in this paper.

3 Alternative Approaches

A review of real-time literature and commercial RTOSs reveals that others have developed approaches that avoid the limitations of fixed-priority interrupt threads. We briefly summarize notable approaches here.

At the heart of each approach is a mechanism for the operating system to track threads pending on interrupt processing. Each approach depends greatly upon the architecture of the operating system and device drivers.

QNX Neutrino [10]. QNX Neutrino is a commercial RTOS with a microkernel architecture. Device drivers are implemented as threaded "servers." Servers receive and execute I/O requests from clients and also perform bottom-half processing. Characteristic to microkernel designs, clients and servers communicate through message passing channels. Device drivers receive requests for I/O operations as messages. In-bound messages are queued, in priority order, in the event that they are sent faster than they can be serviced by the device driver.

In order to avoid priority inversions, device drivers inherit the priority of in-bound messages, which is attached by the sender, when they are *sent*. If messages are queued, then the device driver inherits the maximum priority among queued, and currently processing, messages. In addition to priority, device drivers also inherit the execution time budget of their clients (a mechanism commonly referred to as "bandwidth inheritance"). This allows for the throttling of I/O workloads on a per-client basis.

Bottom-halves are delivered to the device driver for handling as event messages. They are processed at the priority inherited from I/O request messages. Note that privileged threads in user-space may receive and process bottom-half event messages directly for improved efficiency without sacrificing real-time correctness.

Academic microkernels **Credo** [12], an L4 extension, and **NOVA** [11], a microhypervisor, have explored similar techniques. [12] and [11] describe the algorithms used to track priority and budget accounting as messages are passed between threads. This includes interesting cases that may occur when budgets have been exhausted and work has not yet completed.

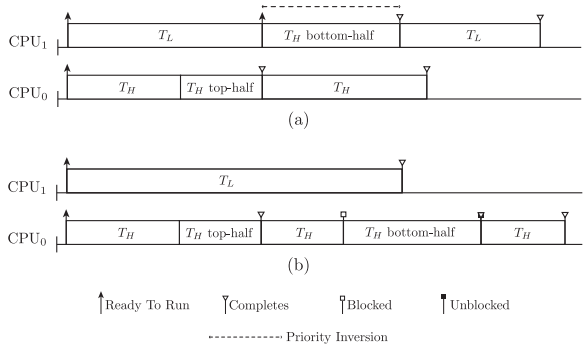

FIGURE 6: *(a) Priority inversions may occur when a bottom-half is co-scheduled with the thread from which it is inheriting priority. (b) Deferring execution of the bottom-half until its owner relinquishes a CPU prevents the priority inversion.*

LynxOS [7] is another microkernel commercial RTOS and it implements message passing methods similar to Neutrino, though bandwidth inheritance does not appear to be supported.

klmirqd [3]. klmirqd is an interrupt handling extension to LITMUS$^{\mathrm{RT}}$, which is itself a real-time extension to Linux. Before a user-task begins an I/O operation, it registers with the OS that it is about to begin using a device, it unregisters when it has completed using the device. Bound to each device is a unique real-time LITMUS$^{\mathrm{RT}}$ kernel thread dedicated to processing bottom-halves from that device. The thread has no scheduling priority when it is idle.

During top-half processing, a registration table is inspected and the identity of the highest-priority thread registered to the interrupt-generating devices is attached to resulting bottom-halves as the "owner" of the bottom-half. The bottom-half is dispatched to the appropriate klmirqd thread, which inherits the priority of the owner of the bottom-half. Bottom-halves may be queued within klmirqd if they are received faster than they are processed; the highest priority queued bottom-half is inherited in such cases. Unlike Neutrino, queues are FIFO ordered.

Unique to klmirqd is the prevention of priority inversions due to asynchronous I/O on globally scheduled multiprocessors. Most real-time analysis techniques assume single threaded workload models. As such, a thread that has its priority inherited by another should never be scheduled simultaneously with that inheriting thread. Otherwise, two threads may be scheduled at the same time under the same "identity" and the non-inheriting thread analytically becomes multi-threaded, breaking analytical assumptions. On a globally scheduled multiprocessor, this may trigger the preemption of another thread that should be scheduled—the preempted thread suffers a priority inversion. For example, in Fig. 6, inset (a), T_L

suffers from a priority inversion when it is preempted by the bottom-half of T_H. The inversion is due to the fact that T_H is already scheduled on the other CPU. In (b), the priority inversion is avoided by deferring the bottom-half until T_H blocks for it to be processed. This scenario is easily possible with asynchronous I/O since the device-using thread may continue to execute while the I/O device performs work. The solution implemented by klmirqd is that the co-scheduling of a bottom-half and its owner is prevented.

Process-Aware Interrupts [13]. Process-Aware Interrupts (PAI) modifies interrupt handling for real-time tasks in Linux. Its primary goal is to reduce overheads due to threaded interrupt handling while still reducing the likelihood of priority inversions.

PAI also uses a registration-based approach like klmirqd to determine the scheduling priorities of bottom-halves. However, instead of dispatching the bottom-half to a thread, one of two actions before the top-half returns the CPU to the interrupted thread: (1) If the priority of the bottom-half is greater than the interrupted thread, then the bottom-half is immediately executed (i.e., the split interrupt handler is merged). (2) Otherwise, execution is deferred and the bottom-half is added to a priority queue. At every context switch, the head of the bottom-half priority queue is checked. If that bottom-half has a higher priority than the next thread selected to be scheduled, then the context switch is aborted and the bottom-half is executed instead. The CPU is rescheduled after the bottom-half has completed.

We found that under worst-case conditions, PAI offers no analytical benefits over *non*-split interrupt handling [3]. However, we also determined that PAI offers very good performance in practice since: (1) bottom-halves are only scheduled when they are *unlikely* to cause priority inversions; and (2) overheads due to threaded interrupt handling are avoided.

4 Conclusion

In this paper we have demonstrated the limitations of fixed-priority interrupt handling in PREEMPT_RT and have briefly surveyed alternative approaches by others that do not suffer from these limitations. Sweeping changes to the device driver and interrupt handling layers of PREEMPT_RT may not be possible due to the colossal size of the existing Linux codebase, so any successful alternatives will likely have to be made incrementally. Nevertheless, we hope this paper serves to facilitate the discussion of viable alternatives that may remedy current limitations.

References

[1] B. Brandenburg, H. Leontyev, and J. Anderson. An overview of interrupt accounting techniques for multiprocessor real-time systems. *Journal of Systems Architecture*, 57(6), 2010.

[2] G. Elliott and J. Anderson. Real-world constraints of GPUs in real-time systems. In *17th Real-Time Computing Systems and Applications*, volume 2, pages 48–54, 2011.

[3] G. Elliott and J. Anderson. Robust real-time multiprocessor interrupt handling motivated by GPUs. In *24th Euromicro Technical Committee on Real-Time Systems*, pages 267–276, 2012.

[4] K. Jeffay and D. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *14th Real-Time Systems Symposium*, pages 212–221, 1993.

[5] M. Lewandowski, M. J. Stanovich, T. P. Baker, K. Gopalan, and A. Wang. Modeling device driver effects in real-time schedulability analysis: Study of a network driver. In *13th Real-Time and Embedded Technology and Applications Systems*, pages 57–68, 2007.

[6] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[7] LynuxWorks, Inc. *Writing Device Drivers for LynxOS*, 2006.

[8] N. Manica, L. Abeni, L. Palopoli, D. Faggioli, and C. Scordino. Schedulable device drivers: Implementation and experimental results. In *6th Operating Systems Platforms for Embedded Real-Time Applications*, pages 53–61, 2010.

[9] W. Mauerer. Àndroit: Real-time for the rest of us. In *Droidcon*, 2012.

[10] QNX Software Systems Limited. *QNX Nutrino RTOS: System Architecture*, 2012.

[11] U. Steinberg, A. Böttcher, and B. Kauer. Timeslice donation in component-based systems. In *6th Operating Systems Platforms for Embedded Real-Time Applications*, pages 16–23, 2010.

[12] U. Steinberg, J. Wolter, and H. Härtig. Fast component interaction for real-time systems. In *17th Euromicro Technical Committee on Real-Time Systems*, pages 89–97, 2005.

[13] Y. Zhang and R. West. Process-aware interrupt scheduling and accounting. In *27th Real-Time Systems Symposium*, pages 191–201, 2006.