Process-Aware Interrupt Scheduling and Accounting *

Yuting Zhang and Richard West
Computer Science Department
Boston University
Boston, MA 02215
{danazh,richwest}@cs.bu.edu

Abstract

In most operating systems, the handling of interrupts is typically performed within the address space of the kernel. Moreover, interrupt handlers are invoked asynchronously during the execution of arbitrary processes. Unfortunately, this allows for a process's time quantum to be consumed by arbitrary interrupt handling. To avoid significant impact to process execution and also to respond quickly enough to interrupts, interrupt servicing is usually split into two parts: a "top" and "bottom" half. The top half executes at interrupt time and is meant to be short enough to complete all necessary actions at the time of the interrupt. In contrast, the bottom half can be deferred to a more suitable point in time to complete servicing of a prior interrupt. Systems such as Linux may defer bottom half handling to a schedulable thread that may be arbitrarily delayed until there are no other processes to execute. A better approach would be to schedule bottom halves in accordance with the priorities of processes that are affected by their execution. Likewise, bottom half processing should be charged to the CPU-time usage of the affected process, or processes, where possible, to ensure fairer and more predictable resource management. This paper describes some of our approaches, both algorithmically and in terms of implementation on a Linux system, to combine interrupt scheduling and accountability. We show significant improvements in predictability of a Linux system by modifying the kernel to more accurately account for interrupt servicing costs and more precisely control when and to what extent interrupts can be serviced.

*This material is based upon work supported by the National Science Foundation under Grant No. 0615153. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

1 Introduction

Over the years there have been numerous operating systems specifically tailored for the predictability requirements of real-time computing. System such as LynxOS [18], QNX [4] and VxWorks as well as many others have been developed for various real-time and embedded computing applications, often demanding low latency and predictable task execution, along with small memory demands. However, part of our work is motivated by the desire to add predictable service management features for application-specific services to off-the-shelf systems that are widely-used, and have relatively low development and maintenance costs. Our current and prior work on "user-level sandboxing" [22] and safe kernel extensions [21] specifically addressed the issue of how to add safe, predictable and efficient application-specific services to commodity OSes, so they could be tailored to the real-time requirements of target applications.

Other research efforts have also investigated the use of commodity operating systems, such as Linux, in real-time environments [10, 11, 12, 15, 16, 19]. RTLinux Free, for example, provides low-latency and predictable execution of kernel-level real-time tasks. It can support tasks with hard real-time guarantees by enforcing bounds on the overheads associated with interrupt processing. This is achieved by modifying the interrupt-handling code within the Linux kernel, so that non-real-time tasks are deferred when there are hard real-time tasks awaiting service. In fact, one of the key observations in RTLinux is that general-purpose systems suffer unpredictability due to potentially unbounded delays caused by interrupt-disabling. Lengthy periods in which interrupts are disabled is a common problem in the critical sections of poorly-written device drivers. However, two other significant problems with general-purpose systems are: (1) interrupt servicing is largely independent of process scheduling, and (2) interrupt accountability is often misrepresented. In the latter case, the time spent servicing an interrupt is usually charged to the process that was ac-

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06 $20.00 © 2006 IEEE

tive at the time the interrupt occurred. The consequence of this is that a process' timeslice is arbitrarily consumed by system-level activities rather than useful work for the real-time task at hand.

The primary goals of this work are essentially two-fold: (1) how to properly account for interrupt processing, so that the time spent in kernel control paths due to interrupts from I/O devices is charged to the appropriate process, where possible, and (2) how to schedule deferrable interrupt handling so that predictable task execution is guaranteed. In addressing these goals, it is important to understand how interrupt processing is dealt with on typical general purpose systems in use today. Systems such as Linux split interrupt service routines into "top" and "bottom" halves, with only the top half requiring execution at the time of the interrupt. Further interrupts may be disabled during a top half, which performs only the basic service requirements at the time of the interrupt (e.g., saving device data to a device-specific buffer and then scheduling a bottom half). Bottom half handling may then be assigned to a schedulable thread, and possibly deferred until there are no other processes to execute. A better approach would be to schedule bottom halves in accordance with the priorities of processes that are affected by their execution. Likewise, bottom half processing should be charged to the CPU-time usage of the affected process, or processes, where possible, to ensure fairer and more predictable resource management.

It should be clear that not all interrupt processing occurs as a result of a specific process request. For example, a hardware timer interrupt may be generated, irrespective of any processes, to update the system clock. However, for I/O devices and corresponding I/O requests (e.g., via `read()` and `write()` system calls on a POSIX system) it is clear that device interrupts are generated on behalf of a specific process. It is for such interrupts that we attempt to prioritize the importance of corresponding bottom half handling. In effect, a bottom half will have a priority consistent with that of the process for which it is performing some service request. Given that we need to determine the target process associated with a given device interrupt, we need to carefully account for the bottom half execution time and then charge that time to the proper process, which may not be the one active at the time of the interrupt.

This paper, therefore, describes some of our methods, both algorithmically and in terms of implementation on a Linux system, to combine interrupt scheduling and accountability. We show significant improvements in predictability of a Linux system by modifying the kernel to more accurately account for interrupt servicing costs and more precisely control when and to what extent interrupts can be serviced. We show how to prioritize interrupt servicing so that the most important real-time processes are assured of making the necessary progress, without suffering system-level CPU time penalties to process interrupts for lower-priority processes.

The remainder of the paper is organized as follows: Section 2 presents the general framework for predictable interrupt accounting and scheduling. The corresponding algorithms for scheduling and accountability are then described in detail in Section 3. Section 4 introduces a simple prototype implementation of our approach in Linux, which is then evaluated experimentally in Section 5. Related work is described in Section 6, after which we discuss conclusions and future work in Section 7.

2 Interrupt Servicing and Accountability Framework

Interrupts are events that can be either synchronous or asynchronous in nature. They are generated by both hardware devices and also by program conditions (i.e., faults, system calls, and traps used for, e.g., debugging purposes). In dual-mode systems such as Linux, that separate the kernel from user-space, it is typical for interrupt-handling to take place in the kernel. Moreover, because interrupts can occur at arbitrary times and usually require at least some basic handling to be performed with "immediate" [1] effect, most systems allow them to be serviced in the context of whatever process is active at the time of their occurrence. However, because many interrupts (e.g., those relating to I/O service requests) occur on behalf of specific processes, it makes sense to service them in accordance with the importance of the corresponding process, or processes. Likewise, proper accounting of CPU usage to handle a specific interrupt should be charged to the process(es) receiving service as a result of the interrupt, not necessarily the process that happens to be executing at the time of interrupt occurrence. With this in mind, we describe our general approach to service interrupts in a "process-aware" manner, showing how it differs from that of the traditional "process-independent" approach typical of general-purpose systems.

Traditional process-independent interrupt service: Figure 1 shows the typical flow of control in general-purposes operating systems resulting from I/O service requests that trigger device interrupts. In this figure, a process issues an I/O request, via a system call (step (1)). The OS then sends the request to the corresponding device via a device driver. At some time later, the hardware device responds by generating an interrupt (step (2)), which is handled by the top half. The deferrable bottom half then completes the necessary service requirements on behalf of the prior interrupt (step (3)), and wakes up the corresponding pro-

[1] In reality, there is a small amount of interrupt dispatch latency to actually save the current execution state and switch to the interrupt handler in question.

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06 $20.00 © 2006 IEEE

cess(es). Since bottom halves may run for a relatively long time (compared to top halves), interrupts are usually enabled during their execution. Therefore, bottom halves can be preempted by the top halves for handling newly occurring interrupts, to ensure fast interrupt responsiveness. Observe that after completion of interrupt handling, a process that is awoken can be rescheduled. Control can then pass back to the user-level process on return from an earlier system call (step (4)).

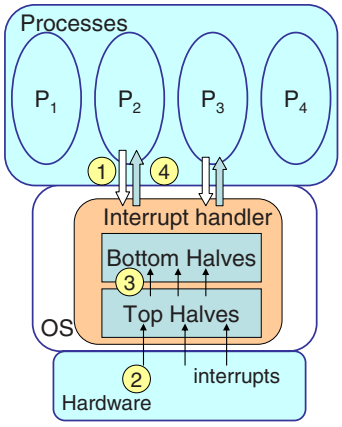

Figure 1. Interrupt handling control path in general purpose systems such as Linux.

Frequently occurring interrupts and, hence, the execution of corresponding top and bottom half handlers may unduly impact the CPU time of a running process. To deal with this scenario, systems such as Linux execute a finite number of pending bottom halves after top half execution, and then defer any subsequently pending bottom halves to a scheduled thread (or until another interrupt occurs, whichever comes first). For example, Linux 2.6 iterates through a function called `do_softirq()` up to 10 times if there are pending "bottom halves" before deferring their execution. Note that we use the term "bottom half" to refer to any deferrable function whose execution is a consequence of a prior interrupt. Linux 2.6 has deprecated the term "bottom half" but still retains deferrable functions in the form of "softirqs" and "tasklets". The deferrable servicing of `softirqs` and `tasklets` in Linux takes place in the context of a task, called `ksoftirqd_CPUxxx` (where `xxx` is a CPU number). However, each of these tasks on different CPUs are given the lowest priority amongst all schedulable tasks and may therefore delay servicing on behalf of another important waiting thread or process. Observe that bottom halves (and, equivalently, softirqs and tasklets) complete the work originated by an interrupt, that may have been generated as a consequence of an I/O request from a process. That process may be blocked, waiting for the completion of the I/O request and, hence, bottom half. If the bottom half's priority does not correspond to that of the blocked process, then the process may wake up too late. This could lead to missed deadlines in a real-time system. Moreover, since interrupts are handled in the context of an arbitrary process, the wrong process may be charged for the time spent handling the interrupt. For the purposes of more accurate CPU accounting, we need to determine the process that is to be awoken, or which is related to the interrupt, and charge its CPU time accordingly.

General-purpose systems such as Linux neglect the relationship between interrupts and corresponding processes, do not properly integrate the scheduling of bottom halves with other processes, and do not accurately account for CPU time usage as a consequence of interrupt servicing.

Process-aware interrupt service: With current hardware support and OS design, we argue that it is reasonable for top halves to execute immediately in order to release I/O devices, since their execution times are meant to be short. However, our approach explores the dependency between interrupts (notably their bottom halves) and corresponding processes, for more predictable service execution and resource accounting. Figure 2 shows the framework of our approach. An interrupt accounting component is added after the execution of bottom halves, to keep track of the processing time of interrupt handlers and to charge this time to the appropriate process(es). An interrupt scheduler is added between the execution of top halves and bottom halves to predict the requesting process and determine exactly *when* bottom halves should execute. We will use the term "interrupt scheduler" to refer to the scheduling of deferrable functions such as bottom halves (or, equivalently, softirqs and tasklets).

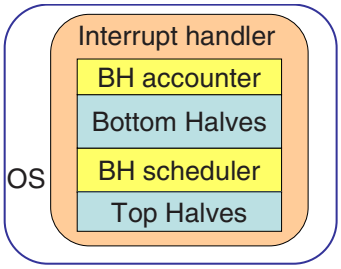

Figure 2. Process-aware interrupt scheduling and accounting framework.

One major challenge is to determine the process associated with a given interrupt and, hence, bottom half. In many cases, the requesting process can only be known at the end of bottom half execution. It is not trivial to find the requesting process before the completion of interrupt processing. For example, consider an interrupt from a network device such as an Ethernet card that indicates the arrival of a packet; on arrival, the packet is processed by a bot-

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06 $20.00 © 2006 IEEE

tom half interrupt handler, to find the proper socket, from which the target process can be determined. Unfortunately, by the time the bottom half completes we may have consumed CPU time in the context of the wrong process and we may have prevented a more important real-time task from executing.

Our approach attempts to *predict*, before the execution of a bottom half, whether or not the corresponding interrupt is for a more important process than the one currently active. If this is the case, we should execute the bottom half immediately after completion of the top half, rather than deferring it in preference for other tasks. Similarly, we need to account for the time spent executing in the bottom half and, once we know the process associated with this bottom half, we should charge it with the CPU time expended in the bottom half. If the target process is not the one active at the time of bottom half processing, then we need to replenish the available CPU time in the current timeslice of the preempted process and deduct the CPU time of the next timeslice from the proper process.

3 Interrupt Servicing and Accountability Algorithms

In this section, we describe the interrupt accounting and scheduling algorithms in detail.

3.1. Interrupt Accounting Algorithm

Based on the current process accounting mechanism, we propose a compensation algorithm for more accurate interrupt and process accounting.

Overview of process accounting mechanism: In most general-purpose systems, such as Linux, time keeping and process accounting is at the granularity of the system clock tick, which is updated according to a hardware timer interrupt. In the timer interrupt handler, the clock time is increased by one tick (i.e., one *jiffy*). On Linux x86 systems, a clock tick typically ranges from $1 - 10ms$, and this one unit of time is charged to the executing process at the current time. It is possible to increase the frequency of timer interrupts and, hence, the precision of the system clock tick, but this comes at the cost of increased interrupt handling overheads [19].

Usually, the execution time of an I/O interrupt handler is less than one clock tick. Fortunately, systems such as Linux leverage hardware support to accurately account for time at granularities smaller than a clock tick. For example, on the x86 architecture there is the `rdtsc` instruction to access the timestamp counter, which is updated at the frequency of the processor and therefore measures time in CPU clock cycles. We use the x86 timestamp counter to measure the time spent executing bottom halves. Knowing the execution time of bottom halves allows us to *compensate* for CPU time usage charged to the wrong process, which just so happens to be preempted while interrupt handling is taking place. Once we determine the process associated with a given interrupt we charge its CPU time usage for the cost of bottom half execution.

Compensation algorithm: There are three steps in the compensation algorithm: (1) measurement of the execution time of a bottom half handler in CPU cycles taken from the average time across multiple bottom halves, (2) measurement of the total number of interrupts processed, and the number processed on behalf of each process, in each clock tick, and (3) adjustment of the system time charged to each process in the timer handler as a result of mischarged interrupt costs. In what follows, we assume that any reference to processing and accounting for interrupts actually applies to bottom half processing on behalf of interrupts. Also, for simplicity, we assume all interrupts and, hence, all bottom halves are for the same device. In practice, the compensation algorithm accounts for the numbers and costs of interrupts for different devices (e.g., interrupts for network devices versus disks).

At system clock time, t, let $N(t)$ be the number of interrupts whose total (bottom half) execution time is one clock tick [2]. Let $m(t)$ be the number of interrupts processed in the last clock tick, and let $x_k(t)$ be the number of unaccounted interrupts for process P_k. Whenever an interrupt is processed, increase m by 1. If process P_j is the process associated with the interrupt, increase $x_j(t)$ by 1. As stated earlier, we determine P_j by the time the bottom half has finished execution.

Suppose the current clock tick is charged to the system execution time of process P_i which is active at time, t. At this point, $m(t)$ is the number of interrupts handled in the context of P_i in the last clock tick. The execution time of these $m(t)$ interrupts is included in this one clock tick, which is possibly mischarged to P_i. Therefore, the difference $m(t) - x_i(t)$ (if positive) is the number of interrupts that have been over-charged to P_i. At each system clock tick, we update the process accounting information in the timer handler as follows:

$x_i(t) = x_i(t) - m(t)$;
$sign =$ sign of $(x_i(t))$;
while $(abs(x_i(t)) >= N(t))$
    system_time(P_i) $+= 1 * sign$;
    timeslice(P_i) $-= 1 * sign$;
    $x_i(t) = x_i(t) - N(t)$;
$m(t) = 0$;

[2] We actually round off $N(t)$ to the nearest integer for cases where the total execution time of $N(t)$ interrupts is not exactly one clock tick.

In the above, system_time(P_i) is the time spent executing at the kernel-level for process P_i in clock ticks. Similarly, timeslice(P_i) is the timeslice of process P_i. To smooth out the short term variabilities of $N(t)$ in each measurement, the current value of $N(t)$ is estimated as an average value $\overline{N(t)}$ at time t using an an Exponentially-Weighted Moving Average (EWMA) with parameter γ ($0 < \gamma < 1$) on the instantaneous values of $N(t)$ as follows:

$$\overline{N(t)} \quad = \quad (1-\gamma) \cdot \overline{N(t-1)} + \gamma \cdot N(t) \qquad (1)$$

Example: Figure 3 gives a simple example to show how the compensation algorithm works. I_i is the interrupt processed on behalf of process P_i. The number of interrupts that need to be compensated on behalf of process P_i at time t is $m(t) - x_i(t)$. Looking at time $t = 1$ in the figure, 3 interrupts have been processed and 2 are for process P_1, so the cost of 1 interrupt should be deducted from the system time charged to P_1. Looking ahead to time $t = 7$, 2 interrupts are charged to P_1 in the last clock tick, although neither were actually for P_1. At this point, 4 interrupts have occurred for P_1 since its last execution and accounting took place. Previously, it was observed that P_1 was over-charged by the cost of 1 interrupt. Consequently, at time $t = 7$, we have $x_1(7) = -2+4-1 = 1$ unaccounted interrupts. When the absolute value of the number of unaccounted interrupts is greater than or equal to $N(t)$ we update the system time and timeslice of the process accordingly.

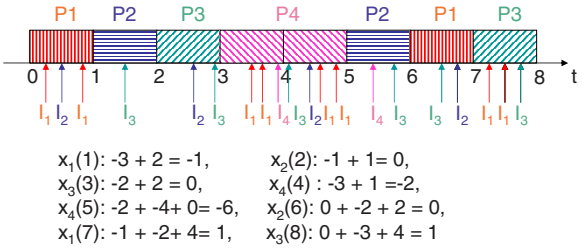

Figure 3. Compensation algorithm example.

3.2. Interrupt Scheduling Algorithm

We propose a process-aware interrupt scheduling algorithm to provide more predictable service execution by scheduling bottom halves. Specifically, the importance of executing a bottom half is dictated by the priority of the corresponding process. The challenge is to identify the corresponding process without having to first execute the bottom half. There are three steps to the algorithm:

- **Finding the candidates:** At the time top half interrupt processing completes, information about the I/O device that raised the interrupt is available. It therefore makes sense that the process which issued a service request that resulted in the interrupt is one of the processes waiting on the corresponding I/O device. This is certainly the case for blocking I/O system calls (e.g., blocking `read()` requests). Moreover, non-blocking requests are not a problem because the process is able to continue its execution and is probably running when the interrupt occurs, unless its timeslice has expired. Notwithstanding, to reduce the overhead of finding candidate processes that could be associated with the interrupt, we require all I/O requests to register the priority of the calling process, and process IDs, with the corresponding device.
- **Predicting the process associated with the interrupt:** With the knowledge of all the candidates, the next step is to infer the process that issued the I/O request and decide the corresponding priority that will be assigned to the bottom half for the interrupt. There are several different prediction schemes we can use. In this paper, we use a simple priority-based predication scheme, in which the bottom half's priority is set to the highest priority of all processes waiting on the corresponding I/O device for a given interrupt. It may be the case that the highest priority process is not the one associated with a specific interrupt occurrence. For example, a packet arriving on an Ethernet device causes an interrupt which is ultimately associated with a low-priority process waiting on a corresponding socket. At the time of the interrupt a higher priority process is also waiting on another socket associated with the same Ethernet device. The justification for our simple highest priority assignment approach is that we typically do no worse than a traditional interrupt processing scheme, in which interrupts are effectively given precedence over all processes, but more often do better at avoiding issues such as priority inversion and incorrect interrupt accounting. Notwithstanding, in the future we will look at more complex predictability models that assess the likelihood of an interrupt being for a given process e.g., by monitoring the history of past interrupts and I/O requests from processes, as well as considering the distribution of priorities of processes waiting on a given device.
- **Scheduling the bottom half:** After predicting the priority of the process associated with the device interrupt, the interrupt-level scheduler decides whether it is more important to service the bottom half or the process active at the time of the interrupt. The bottom half is given the priority of the process associated with the interrupt for the purposes of scheduling decisions. If deemed more important, the bottom half executes without delay, otherwise it is deferred until the next scheduling point.

In our approach, the interrupt scheduler is executed whenever the bottom halves are activated and ready to be executed. To avoid unnecessary context switch overhead, we also invoke the interrupt scheduler in the process scheduler. Before an actual context switch to a newly selected process, the interrupt scheduler compares the priority of the next process to that of the bottom halves. If next process has higher priority, the actual context switch is performed. If any bottom halves have higher priority, they are executed instead without the process context-switch taking place.

Example: Figure 4(a) shows an example time-line for scheduling a bottom half in the presence of 2 background processes in Linux. At time t_1, P_1 sends an I/O request, and is blocked while waiting for the request to complete. At this point, the process scheduler selects P_2 for execution. In the context of P_2, at time t_2, the I/O device responds to the I/O request by P_1 , and raises an interrupt. The corresponding top half (labeled It_1) is activated and later finishes at time t_3. At some time, t_4, the bottom half (labeled IB_1) is executed, and it finishes at time t_5. Finally, P_1 is woken up and eventually scheduled at time t_6. Both the execution time of top and bottom halves is charged to P_2. However, with our compensation algorithm, we will be able to charge the execution time of the bottom half to process P_1 as shown in Figure 4(b).

In this paper, we focus on the delay time between t_3 and t_4 which is affected by the interrupt scheduler. Observe that if the time between t_4 and t_3 is small P_2's progress may be unduly affected. Moreover, if P_2 happens to be more important than P_1 we end up delaying the progress of P_2 on behalf of a bottom half for a lower priority process. Worse still, if P_2 happens to be a real-time process with a deadline, the execution of a bottom half could cause P_2 to miss that deadline. If the opposite action is taken, so that the bottom half is deferred until no other processes are ready to execute, we may unduly impact the progress made by P_1 awaiting the outcome of its I/O request. In our approach, if P_1 has higher priority, we want its bottom half to be executed as soon as possible as in Figure 4(b). However, if P_2 has higher priority, we want P_2 to continue to execute until it finishes, then to execute P_1's bottom half as shown in Figure 4(c).

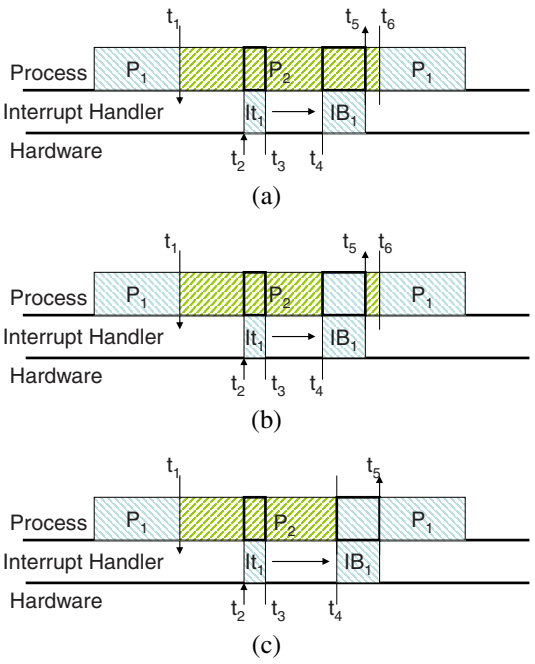

Figure 4. Alternative approaches to scheduling bottom halves versus processes.

4 System Implementation

Our prototype implementation only requires a few minor modifications to the base Linux kernel. As a case study, we focus on interrupt handling associated with network packet reception, and implement the scheduling and accounting component of our framework on top of the existing bottom half mechanism in Linux. Linux uses `softirqs` to represent bottom halves. To be consistent, we refer to softirqs and bottom halves interchangeably in the following text.

Control path for packet reception in the Linux kernel: Figure 5 shows the control path of system calls for receiving UDP packets in Linux. A socket structure is used to send or receive packets in Linux, and each socket can be associated with a specific IP address and port number using `bind()` and `connect()` system call semantics. Other system calls such as `read()`, `recvmsg()` and `recv()` can be made by applications to receive data on a given socket. The kernel checks whether there are any packets in the receiving queue for the specified socket and, if so, they are copied to a target user-level process. Since the kernel cannot predict when a packet will arrive on a network interface, the networking code that takes care of packet reception is executed in top and bottom half interrupt handlers.

At some future time, when a packet carrying the right hardware address arrives at the network device, it is stored in a device buffer and then the device raises an interrupt. The device-specific interrupt handler (i.e., the top half "irq handler") copies the packet from device memory to a kernel socket buffer, does other necessary operations, and then enables the `NET_RX_SOFTIRQ` (bottom half for network packet reception) to be scheduled at some convenient time. When `do_softirq()` is executed to schedule bottom halves, it checks whether there are pending `NET_RX_SOFTIRQ`s, and calls the corresponding bottom half handler `net_rx_action()` to receive all arrived packets if there are any. The function `netif_receive_skb()` is the main code to re-

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06 $20.00 © 2006 IEEE

ceive a packet. It in turn ultimately invokes a suitable function for the network layer protocol in operation, such as `udp_rcv()` to receive UDP packets. The target socket is determined, and any received packets are appended to that socket's receiving queue. Finally, `sock_def_readable()` is invoked to essentially wake up a process sleeping in the socket's wait queue. When the application process is woken up it continues the path of a particular protocol (here, `udp_recvmsg()`), which copies each packet payload into a user-space buffer [1].

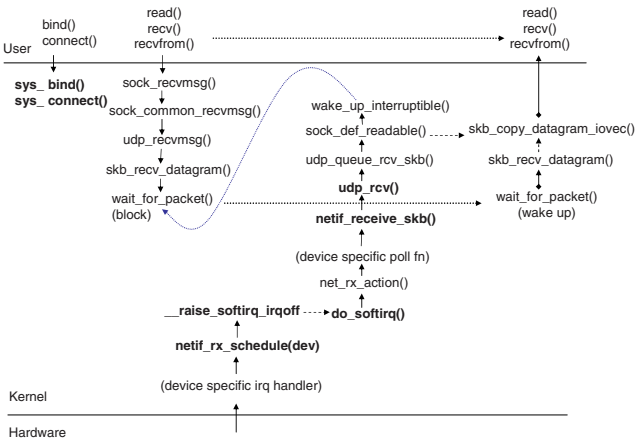

Figure 5. Control path for receiving UDP packets in the Linux kernel.

Implementation of the interrupt accounting algorithm: The execution time of the softirq handler `net_rx_action()` is measured for each invocation, using the high-precision Pentium-based Timestamp Counter (TSC). More than one packet may be received in each handler invocation. By measuring the number of packets received, and the total time to process these packets in bottom halves, we are able to calculate the number of packets, N, whose total execution time is up to one clock tick. The EWMA value of $N(t)$, $\overline{N(t)}$ (as defined in Section 3) is used to compensate processes whose system time is consumed by handling interrupts for other processes. We use a γ value of 0.3 to calculate the EWMA. When the proper socket is found for the packet (e.g. in `udp_rcv()`), the right process for the packet is known. This enables us to keep track of the number of packets received for each process, as well as the total number of packets received from the network device in each clock tick. We also keep track of the number of orphan packets whose requesting processes are unknown. The essential part of service time compensation is implemented in the timer handler. Specifically, we modify the `account_system_time()` kernel function to update the system time accounting and timeslice of the current process at the granularity of one clock tick, according to the algorithm in Section 3.1.

Implementation of the interrupt scheduling algorithm: The interrupt (bottom half) scheduler is mainly implemented in the `do_softirq()` function. Before the corresponding softirq handler (e.g. `net_rx_action()` for `NET_RX_SOFTIRQ`) is invoked, it checks the eligibility of each softirq, and decides whether to schedule it based on the priority of the softirq versus the priority of the current process. From the control path in Figure 5, we can see that before the softirq is executed, the right device for the current softirq is known. Thus all related sockets for the network device, and all processes sleeping on the sockets, are known. These relationships can be established explicitly when the processes issue `bind()/connect()` and `read()/recv()` system calls, so the overhead of finding this information in the bottom half scheduler is reduced. Then, based on the priority of the candidate processes waiting for a network device interrupt, along with the prediction algorithm used to determine which process the current interrupt is likely to be for, we can determine the priority of each softirq at each scheduling point.

5 Experiments

Using our prototype implementation, we conducted a series of experiments to evaluate the performance of the proposed approach for interrupt scheduling and accountability, focusing on network packet reception interrupts. We used a UDP-server process running on an experimental machine, which simply binds to a port, and receives arriving packets for the specified port. A UDP-client process on another machine keeps sending UDP packets to the port of the experimental machine specified by the UDP-server process. The interrupt arrival rate is adjusted by varying the packet sending rate of the UDP-client. To evaluate the interrupt effect on other processes, we run another CPU-bound process on the experimental machine concurrently with the UDP-server process.

Both UDP-server and UDP-client machines have 2.4GHz Pentium IV processors with 1.2GB of RAM. They are connected by gigabit Ethernet cards, and each machine runs Linux-2.6.14 with 100 Hz timer resolution. We compare the performance of interrupt and process servicing in the original Linux-2.6.14 kernel with that of a modified Linux-2.6.14 kernel, patched with our process-aware interrupt scheduling and accounting components on the server machine.

Accounting accuracy: In this set of experiments, we set the CPU-bound process to be a real-time process with priority 50 in the `SCHED_FIFO` class, while the UDP-server is a non-real-time process. Therefore, the CPU-bound process always preempts the UDP-server process, and runs for its entire timeslice when it is ready to execute. The CPU-

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06 $20.00 © 2006 IEEE

bound process simply loops through a series of arithmetic operations, without making any system calls, so that its execution time should be spent at user-level. We record the number of *jiffies* that each process spends executing at user-level versus system- (or kernel-) level, by reading from `/proc/pid/stat` for each process with a specific `pid`. Each jiffy represents 10ms. The results are averaged over 5 trials of each experiment. In each trial, the CPU-bound process runs for 100s and then sleeps for 10s. During the whole 110s period, the UDP-client keeps sending packets of 512 bytes data to the UDP-server at a constant rate. Since the CPU-bound process has a relatively high priority, the UDP-server can only run after each period of 100s when the CPU-bound process goes to sleep. Thus, network interrupts are processed in the context of the CPU-bound process in the first 100s of each 110s period (unless they are deferred to their own threads, which is not the case in our experiments).

Figure 6(a) compares the accounted execution time of the CPU-bound process in Linux (labeled "Linux") against the accounted time in our modified Linux system (labeled "Linux-IA" for Interrupt Accounting). For completeness, we also show the optimal value (labeled "Opt"), which represents how much time should be accounted for the CPU-bound process executing entirely at user-level every 100s. Any system-level time charged to the CPU-bound process is due to interference from interrupts such as those associated with the the network interface. Ideally, we do not want any interference from network interrupts associated with the UDP-server process, because that process is set to a distinctly lower priority so that we can guarantee the predictability of the real-time CPU-bound process. However, given that we have to account for system-level interrupt handling somewhere, we wish to ensure that the CPU-bound process is not unduly charged for unrelated system-level activities.

In Figure 6(a), the x-axis is the packet sending rate from the UDP-client. As the sending rate increases, the UDP-server machine observes a correspondingly higher rate of network interrupts and, hence, more time spent servicing top and bottom halves. Thus, the user-level CPU time consumed by the CPU-bound process decreases significantly, as shown by the `Opt` value. However, in the original Linux case, all 100s of time during which the CPU-bound process is active are charged to the CPU-bound process, along with time spent handling network interrupts. In contrast, with our interrupt accounting component, the time consumed by interrupt handling is more accurately compensated to the CPU-bound process and, instead, charged to the UDP-server process. As can be seen, the accounted time in the `Linux-IA` case is very close to the optimal value.

Figure 6(b) shows the ratio between the accounting error and the optimal value, calculated as $\frac{t-t_o}{t_o}$, where t is the actual accounted time for the execution of the CPU-bound process every 100s, while t_o is the equivalent optimal value discarding the interrupt servicing time. As can be seen, the error in accounting accuracy is as high as 60% in the original Linux kernel, while it is less than 20% (and more often less than 5%) in our `Linux-IA` approach.

Figure 6(c) shows the absolute value of compensated time for both the CPU-bound process and the UDP-server in our `Linux-IA` approach. The bars for the "UDP-server (a)" case represent the time charged (due to interrupts) to the UDP-server in the first 100s of every 110s period, when the CPU-bound process is running, while the bars for the "UDP-server(b)" case represent the time charged (again, due to interrupts) over the whole 110s period. We can see that the time compensated to the CPU-bound process is accurately charged to the UDP-server process in the first 100s period. In the last ten seconds of each period, the high priority CPU-bound process is sleeping, allowing the UDP-server to execute and be correctly charged for interrupt handling.

Bottom half scheduling effects: Using the same scenario as above, we evaluate the scheduling impact of bottom halves on the CPU-bound process and the UDP-server process using our modified Linux kernel versus an unmodified kernel. In this case, our modified kernel is patched with *both* interrupt accounting and scheduling components.

Figure 7(a) compares the user-time consumed by the CPU-bound process running on a vanilla Linux system (labeled "Linux") versus our patched system (labeled "Linux-ISA" for Interrupt Scheduling and Accounting). As explained before, for the original Linux kernel, the user-time consumed by the CPU-bound process in a given real-time period (here, 100s) decreases when the packet sending rate from the UDP-client machine increases. However, for the `Linux-ISA` case, the CPU-bound process is not affected by network interrupt handling. In fact, the CPU-bound process can still use up to 100% of time during the first 100s of each period when it is running. In the `Linux-ISA` case, the priority of the interrupt is set to that of the requesting process, which is the UDP-server. Since the UDP-server has lower priority than the CPU-bound process, the network bottom half handler cannot preempt the CPU-bound process. Therefore, the predictable execution of the CPU-bound process is guaranteed.

Figure 7(b) shows the time consumed by interrupt handling on behalf of the UDP-server over each $110s$ interval. In the `Linux-ISA` case, network bottom half handling only starts to consume CPU time after the CPU-bound process goes to sleep, whereas the `Linux` case incurs significant interrupt handling costs during the time the CPU-bound process is runnable. Figure 7(c) shows the packet reception rate of the UDP-server, which is the ratio between the number of packets received in UDP-server to the number of packets sent by the client. It can be seen that the

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06 $20.00 © 2006 IEEE

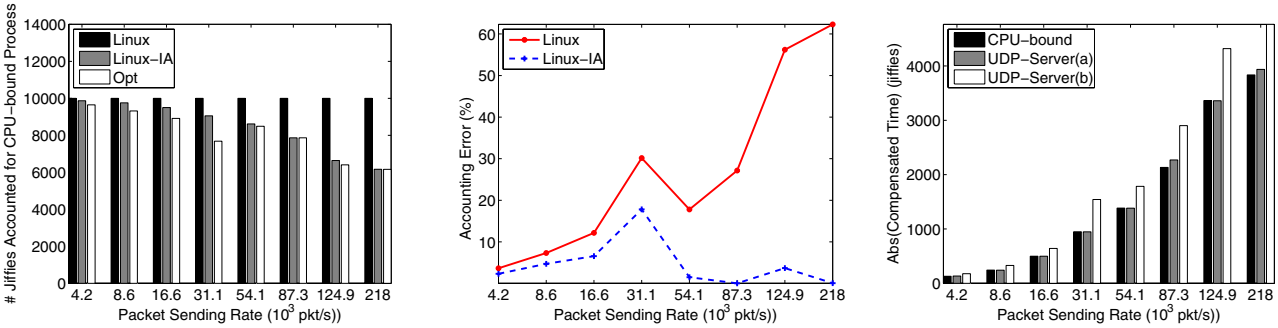

Figure 6. CPU time accounting accuracy.

receiving rate in the `Linux-ISA` case is very close to that of the original Linux kernel. Both are less than 10%. In the unpatched Linux case, most packets cannot be received by the user-level UDP-server until the CPU-bound process sleeps, due to their relative priorities. Therefore, the time spent receiving packets into kernel socket buffers during bottom half handling in the unpatched Linux case is largely wasted. This is partly because they may be over-written before the UDP-server reads them. Consequently, our Linux interrupt scheduling and accounting framework can guarantee the predictable execution of processes with high priorities, while not degrading the service of processes with lower priorities.

As a final experiment, the UDP-client sends bursts of packets according to a two-state Markov Modulated Poisson Process (MMPP-2), with average geometric burst sizes of 5000 packets, and different average exponential burst inter-arrival times. The CPU-bound process is set as a periodic real-time process with a constant period of 1s in which it must execute for 0.95s. The end of each 1s period acts as a deadline. From Figure 8(a), there are no missed deadlines in `Linux-ISA`. The service of the real-time process is guaranteed, and the interrupt handling is deferred until the CPU-bound process finishes execution in its current period. This contrasts with the high deadline miss rate in the `Linux` case as the packet sending rate increases. As can be seen from Figure8(b), interrupts consume a lot of time during the execution of the CPU-bound process. However, the time spent handling the packet reception interrupts does not help the performance of the UDP-server in the original `Linux` case, as shown in Figure 8(c). The packet reception rate is even less than in `Linux-ISA`, and down to 0% when the sending rate is high. This is because the CPU-bound process cannot finish its execution before the start of its next period due to the costs of interrupt handling, and thus is always competing for CPU cycles. As the lower priority process, the UDP-server never has the opportunity to wake up and read its packet data. The unpredictable interrupt service not only affects the CPU-bound process, but also the UDP-server. However, in `Linux-ISA`, the packet reception rate is pretty constant. With guaranteed service of the CPU-bound process, the UDP-server is able to be scheduled to receive packets after the CPU-bound process finishes its execution in each period. `Linux-ISA` improves service to all processes on account of its interrupt scheduling.

6 Related Work

A number of systems have added real-time capabilities to off-the-shelf time-sharing kernels to provide both predictability and applicability for a large application base [19, 10, 20, 16, 15]. Some researchers have focused on the structure of systems and the use of micro-kernels in real-time application domains, showing how the communication costs between separate address spaces is often less than the overheads associated with interrupt processing [9]. Other research efforts such as RTLinux and RTAI, for example, provide support for hard real-time tasks by modifying a base Linux kernel to essentially intercept the delivery of interrupts. Such approaches ensure that legacy device drivers cannot disable interrupts for lengthy periods that affect the predictability of the system. None of these systems, however, have focused on the scheduling and accounting of deferrable bottom halves in an integrated manner, to ensure predictable service guarantees on commercial off-the-shelf systems.

The traditional interrupt-handling mechanism in existing off-the-shelf operating systems is largely independent of process management, which can compromise the temporal predictability of the system. The idea of integrating interrupt handling with the scheduling and accountability of processes has been investigated to different degrees in several research works. Kleiman and Eykholt [6] treat interrupts as threads, which allows a single synchronization model to be used throughout the kernel. However, interrupt threads still use a separate rank of priority levels separate from those of processes, whereas we associate the priorities of bottom halves with corresponding processes. By

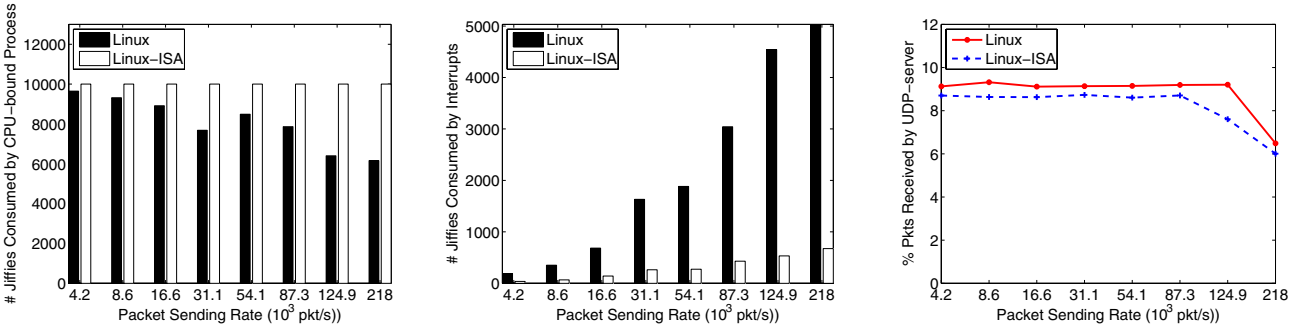

Figure 7. Network bottom half scheduling effects.

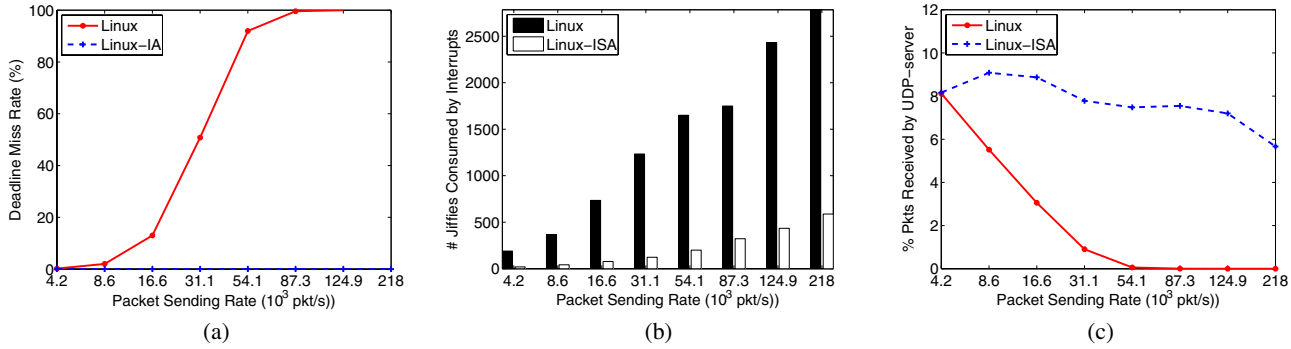

Figure 8. Deadline miss rates and other performance characteristics under bursty traffic conditions.

contrast, Leyva-del-Foyo et al proposed a unified mechanism for synchronization and scheduling of both interrupts and processes [8]. In this work, interrupts are assigned dynamic priorities and scheduled like processes (or tasks) within their own contexts. In the absence of special hardware support, to integrate the scheduling of tasks and interrupts, a software-based abstraction layer must strategically mask the delivery of certain interrupts to ensure task predictability. This whole approach may, however, yield increased context-switching overheads to service interrupts.

There are other alternative techniques to integrate interrupt and process scheduling without introducing a separate context for interrupt handling. Regehr et al[13, 14] describe a hierarchy of software and hardware schedulers to handle the overload caused by interrupts. By using well-known scheduling techniques such as sporadic servers, the service for interrupts is constrained to prevent overload. Similarly, Facchinetti et al [3] propose a hierarchical scheduling framework with an assigned bandwidth to interrupts, so that the real-time application tasks can be guaranteed service that is independent of the interrupt server. However, the interrupt policy is completely independent from the scheduling policy for the application processes. In other work, Jeffay and Stone [5] analyzed the schedulability of tasks while considering interrupts as the highest priority class of events. None of the above research works explore the dependency between interrupts and processes, and use this information to decide the priority of interrupts (essentially bottom halves) in CPU scheduling. The novelty of our work is based on the combined scheduling and accountability of interrupts associated with corresponding processes that issue service requests on I/O devices.

The issue of scheduling dependencies has been addressed in many other areas of research over the years. Priority inheritance [7] and priority ceiling protocols [17], for example, were proposed to schedule processes with resource dependencies. Clark's work on DASA [2] focused on the explicit scheduling of real-time processes by grouping them based on their dependencies. In these bodies of work there is an assumption that dependencies are known in advance before a process is scheduled. However, Zheng and Nieh [23] developed the SWAP scheduler that automatically detects process dependencies and accounts for such dependencies in scheduling. In contrast to these research works, we address the dependencies between interrupts and processes, rather than those amongst only processes. By using various prediction techniques, we are able to more accurately schedule and account for interrupts, so that other

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06 $20.00 © 2006 IEEE

processes in the system are assured of their timeliness requirements.

7 Conclusions and Future Work

In this paper, we describe a process-aware interrupt scheduling and accounting framework for general-purpose systems such as Linux. Our approach focuses on the accountability and scheduling of bottom halves associated with I/O interrupt handling. By exploring the dependency between device interrupts and processes, we show how to properly account for interrupt handling and how to schedule deferrable bottom halves so that predictable task execution is guaranteed.

Our approach attempts to predict the importance of an interrupt, based on the priorities of process(es) waiting on the device generating that interrupt. Given this prediction, our interrupt scheduling scheme decides whether to execute or defer a bottom half by comparing its priority to that of the currently executing process. In addition, our scheme accounts for the time spent executing a bottom half and properly adjusts the system time and timeslice usage of affected processes using a compensation algorithm.

This paper describes our approach both algorithmically and in terms of implementation on a Linux system, to combine interrupt scheduling and accountability. From the experiment results on the prototype implementation of our approach in Linux systems, we show significant improvements in accountability and predictability of interrupt servicing. By properly prioritizing the interrupts, the most important real-time processes are assured of making the necessary progress, without suffering system-level CPU time penalties to process interrupts for lower-priority processes.

Our current implementation only uses a relatively simple priority-based prediction scheme to derive the priority of an interrupt's bottom half. In future work, we will investigate other prediction schemes, to derive the priorities of bottom halves without having to execute them first in order to determine the corresponding process(es). We are also interested in analyzing the performance of the proposed approach integrated with other real-time process scheduling algorithms, such as our earlier window-constrained algorithms. Finally, we intend to validate our interrupt scheduling and accountability framework in embedded real-time systems.

References

[1] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel, Second Edition*. O'Reilly & Associates, Inc., 2002.

[2] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, 1990.

[3] T. Facchinetti, G. Buttazzo, M. Marinoni, and G. Guidi. Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems. In *17th Euromicro Conference on Real-Time Systems*, 2005.

[4] D. Hildebrand. An architectural overview of QNX. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, 1992.

[5] K. Jeffay and D. L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, December 1993.

[6] S. Kleiman and J. Eykholt. Interrupts as threads. *ACM SIGOPS Operating Systems Review*, 1995.

[7] B. Lampson and D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 1980.

[8] L. E. Leyva-del-Foyo, P. Mejia-Alvarez, and D. de Niz. Predictable interrupt management for real time kernels over conventional PC hardware. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, San Jose, CA, USA, April 2006.

[9] F. Mehnert, M. Hohmuth, and H. Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, Austin, Texas, December 2002.

[10] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in Linux. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium (RTAS)*, June 1998.

[11] QLinux: http://www.cs.umass.edu/ lass/software/qlinux/.

[12] RED-Linux: http://linux.ece.uci.edu/red-linux/.

[13] J. Regehr and U. Duongsaa. Eliminating interrupt overload in embedded systems. Unpublished.

[14] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, December 2003.

[15] Real-Time Application Interface: http://www.rtai.org.

[16] Real-Time Linux: http://www.rtlinux.org.

[17] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 1990.

[18] V. Sohal and M. Bunnell. A real OS for real time - LynxOS provides a good portable environment for embedded applications. *Byte Magazine*, 21(9):51, 1996.

[19] 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 *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium (RTAS)*, June 1998.

[20] H. Tokuda, T. Nakajima, and P. Rao. Real-time Mach: Towards a predictable real-time system. In *Proceedings of USENIX Mach Workshop*, pages 73–82, 1990.

[21] R. West and J. Gloudon. 'QoS safe' kernel extensions for real-time resource management. In *the 14th EuroMicro International Conference on Real-Time Systems*, June 2002.

[22] R. West and G. Parmer. Application-specific service technologies for commodity operating systems in real-time environments. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, San Jose, California, April 2006.

[23] H. Zheng and J. Nieh. Swap: A scheduler with automatic process dependency detection. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI-2004)*, March 2004.