# Virtual-CPU Scheduling in the Quest Operating System

Matthew Danish, Ye Li and Richard West

Computer Science Department
Boston University
Boston, MA 02215, USA
{md,liye,richwest}@cs.bu.edu

## Abstract

*This paper describes the scheduling framework for a new operating system called "Quest". The three main goals of Quest are to ensure safety, predictability and efficiency of software execution. For this paper, we focus on one aspect of predictability, involving the integrated management of tasks and I/O events such as interrupts. Quest's scheduling infrastructure is based around the concept of a virtual CPU (VCPU). Using both Main and I/O VCPUs, we are able to separate the CPU bandwidth consumed by tasks from that used to complete I/O processing. We introduce a priority-inheritance bandwidth-preserving server policy for I/O management, called PIBS. We show how PIBS operates with lower cost and higher throughput than a comparable Sporadic Server for managing I/O transfers that require small bursts of CPU time. Using a hybrid system of Sporadic Servers for Main VCPUs, and PIBS for I/O VCPUs, we show how to maintain temporal isolation between multiple tasks and I/O transfers from different devices. We believe Quest's VCPU scheduling infrastructure is scalable enough to operate on systems supporting large numbers of threads. For a system of 24 Main VCPUs, we observe a CPU scheduling overhead of approximately* $0.3\%$ *when VCPU budget is managed in* $1ms$ *units.*

## 1 Introduction

Low latency and predictable task execution is fundamental to the design of a real-time operating system. However, the complex interactions between the various flows of control within a system pose significant challenges in terms of meeting those low latency and predictability requirements. For example, asynchronous events caused by interrupts generated by I/O devices may interfere with the timely execution of tasks. Deadlocks, starvation, priority inversion and synchronization issues all add to the difficulty of ensuring real-time execution guarantees. Additional areas of unpredictability include paging activity in virtual memory systems, blocking delays due to synchronous access to shared resources, unaccounted time spent in different services (including the task scheduler and dispatcher), and crosstalk [4] between the execution of different control flows that impacts shared resources such as caches.

While there are numerous operating systems that have either been designed purposely for real-time computing (e.g., LynxOS [22], QNX [8], and VxWorks), or have been extended from off-the-shelf technologies (e.g., RTLinux [20], and RTAI [19]), these systems can still suffer from unpredictability and timing violations due to lack of *temporal isolation* between tasks and system events. Here, temporal isolation refers to the property of ensuring tasks receive guaranteed resource allocations (e.g., minimum CPU cycles) over specific windows of real-time, even in the presence of other executable threads. If necessary, certain threads of control such as those associated with interrupt handling must be deferred from immediate execution, or denied certain resources, to guarantee the timely execution of other tasks.

In an attempt to provide temporal isolation and improved system predictability, we modified Linux to support a series of bandwidth preserving servers based on Deferrable and Sporadic Server policies [23, 25]. Rather than changing existing APIs, such as POSIX, so that explicit time-constraints are placed on system service requests, we designed an architecture based around the notion of a "virtual CPU" (VCPU). Each virtual CPU is assigned a specific share of physical CPU (PCPU) resources and operates as a bandwidth preserving server for all the threads it supports. As we progressed with our Linux developments, we concluded that it would be easier to design a new operating system rather than retrofit a system that is not fundamentally designed to be real-time. For this reason, we set about developing an entirely new OS called "Quest", featuring a VCPU scheduling framework that is the focus of this paper.

In common with existing approaches to implement rate-limiting servers, each VCPU in Quest has a *budget* and *replenishment period*, making the system amenable to real-time analysis applicable to traditional periodic task models. In our approach, system designers and application developers would use APIs to create and destroy VCPUs, define time-constraints on their execution, control the means by which they are bound to physical CPUs using affinity masks, and to establish scheduling classes for the association of VCPUs with tasks.

One of the motivations for the VCPU model is our earlier work on process-aware interrupt scheduling and accounting [29]. By associating the importance of executing an interrupt "bottom half" handler with the priority of a blocked task waiting on the corresponding I/O device, we showed how to achieve greater predictability for real-time tasks (or threads) without undue interference from interrupts on behalf of lower-priority tasks [1]. However, our earlier work did not bound the amount of time spent executing interrupts and livelock [14] remained an issue. Using special I/O VCPUs with appropriately chosen scheduling parameters (including service budgets), it is possible to schedule bottom half interrupt handlers without incurring livelock. Consequently, our scheme proposes two types of VCPUs: Main VCPUs are associated with normal thread execution, while optional I/O VCPUs are associated with (I/O-based) interrupt scheduling and accounting on behalf of threads blocked on their Main VCPUs (see Figure 1 for further details). In this situation, when a thread waiting on I/O is unblocked, it becomes eligible for execution on its Main VCPU once again.

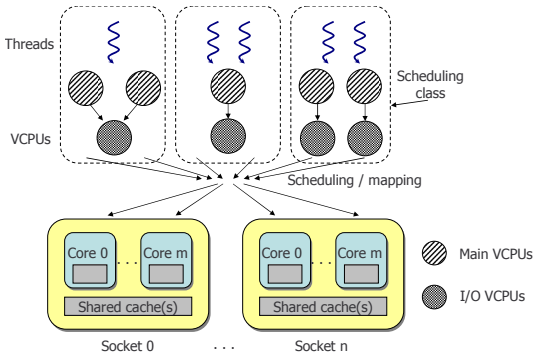


**Figure 1. VCPU Scheduling in Quest**

I/O VCPUs make it possible to separate the processing capacity for I/O events from that assigned to tasks. The flexibility of Quest allows for an I/O VCPU to be shared by multiple threads associated with one or more Main VCPUs, each having different priorities. Here, it is possible for the I/O VCPU to inherit the priority of the Main VCPU from which an I/O request originated and on whose behalf I/O event processing is taking place. We show how this approach alleviates the need for separate I/O VCPUs for each task issuing I/O requests, thereby reducing the number of VCPUs in the system and the total overhead that would otherwise be required to manage them.

We begin the following section by describing the Quest VCPU architecture in more detail, and the rationale for various design decisions. Experimental results are then described in Section 3. This is followed by a discussion of related work, and finally, conclusions and future work.

## 2 The Quest Operating System

Quest currently operates on 32-bit x86 architectures, and leverages hardware MMU support to provide page-based memory protection to processes and threads. As with UNIX-like systems, segmentation is used to separate the kernel from user-space. In contrast to existing systems, we are considering memory protection techniques based on fine-grained fault-isolation around software components. Ideas similar to those in our prototype work on Composite [17], which uses Mutable Protection Domains (MPDs) [16] are being considered for safety in Quest.

Quest is a SMP system, operating on multicore and multiprocessor platforms. It has support for kernel threads, and a network protocol stack based on "lightweight IP" (lwIP) [12]. The source tree is approximately 175 thousand lines of code, including drivers and lwIP. However, the core kernel code is approximately 11 thousand lines. A performance monitoring subsystem is being developed, to inspect hardware performance counters available on modern processors, for efficient micro-architectural resource management. For example, some of our work is being used to gather shared cache information using performance events that record cache misses and hits to estimate cache occupancies [27]. This information can be used to improve co-runner selection decisions on chip multiprocessors (CMPs), to reduce cache conflict misses and expensive memory stall costs caused by cache-intensive workloads.

### 2.1 VCPU Scheduling Subsystem

Of particular interest to this paper is the scheduling subsystem, and the rationale for its design. In Quest, VCPUs form the fundamental abstraction for scheduling and temporal isolation of the system. The concept of a VCPU is similar to that in virtual machines [2, 4], where a hypervisor provides the illusion of multiple PCPUs represented as VCPUs to each of the guest virtual machines. While Quest is not a hypervisor geared towards hosting guest virtual machines, we use VCPUs for scheduling and accounting CPU resources on behalf of multiple threads. VCPUs exist as

[1] A "bottom half" refers to a deferrable portion of an interrupt handler.

kernel abstractions to simplify the management of resource budgets for potentially many software threads. We use a hierarchical approach in which VCPUs are scheduled on PCPUs and threads are scheduled on VCPUs.

In common with bandwidth preserving servers, each VCPU, $V$, has a maximum compute time budget, $C_{max}$, available in a time period, $V_T$. $V$ is constrained to use no more than the fraction $V_U = \frac{C_{max}}{V_T}$ of a physical processor (PCPU) [2] in any window of real-time, $V_T$, while running at its normal (foreground) priority. To avoid situations where PCPUs are otherwise idle when there are threads awaiting service, a VCPU that has expired its budget may operate at a lower (background) priority. All background priorities are set distinctly below those of foreground priorities to ensure VCPUs with expired budgets do not adversely affect those with available budgets. Under this model, a VCPU simply serves as a resource container [3] for the scheduling and accounting of PCPU resources shared amongst competing software threads.

Figure 1 shows the arrangement of VCPUs, PCPUs (i.e., cores) and threads. Threads are constrained to use VCPUs within their scheduling class, which contains at least one Main VCPU and zero or more I/O VCPUs depending on the need for I/O resources. Having a distinction between main and I/O VCPUs is motivated by several factors: (1) separate bandwidth constraints can be placed on I/O processing and task execution, and (2) I/O VCPUs can be configured to serve I/O requests from individual or groups of tasks (similarly, for individual or groups of devices).

As an example, consider two Main VCPUs, $V_1$ and $V_2$ with bandwidth factors $V_{U,1}$ and $V_{U,2}$, respectively, while an I/O VCPU, $V_{IO}$, has a separate (perhaps lower) bandwidth factor $V_{U,IO}$. $V_{IO}$ may be shared by both Main VCPUs or it may be restricted to just one of them, depending on the classification of VCPUs. Similarly, $V_{IO}$ may be configured to service multiple I/O devices or just one. Quest allows full flexibility in this regard, including the ability of a thread running on one Main VCPU to access separate I/O devices through different I/O VCPUs. Suppose now that a task, $\tau_1$, associated with $V_1$ issues an I/O request to some device. In the absence of $V_{IO}$ the system would require that I/O processing be performed using $V_1$. Given that CPU capacity available to $V_1$ may be far greater than that available to $V_{IO}$, it is possible that a high rate of I/O events could consume a burst of CPU budget, thereby preventing tasks from execution. Directing the management of I/O to $V_{IO}$ and limiting CPU usage to a lower level than that available to $V_1$ ensures less interference to task execution. It should be noted here that while $\tau_1$ is awaiting completion of an I/O request, $V_1$ could be assigned to another task.

[2]Unless otherwise stated, a PCPU may be a uni-processor, or a core/hardware thread of a multicore CPU.

**Main VCPUs.** In Quest, Main VCPUs are by default configured as Sporadic Servers. We use the algorithm proposed by Stanovich et al [25] that corrects for early replenishment and budget amplification in the POSIX specification. Fixed priorities are used rather than dynamic priorities (e.g., associated with deadlines) so that we can treat the entire system as a collection of equivalent periodic tasks scheduled by a rate-monotonic scheduler (RMS) [11]. Rate-monotonic analysis can then be used to ensure the utilization bound on any single PCPU does not exceed that required for a feasible schedule. In this approach, priorities are set inversely proportional to VCPU periods.

While a scheduling class defines a collection of threads and VCPUs, it is possible to assign different priorities to VCPUs (and also threads) within the same class. Moreover, multiple threads within the same class may share one or more VCPUs for their execution. By defaulting to a fixed priority scheme for scheduling VCPUs, we avoid the overhead associated with updating priorities dynamically, as would be the case if VCPUs had associated deadlines. While the least upper-bound on utilization for feasible schedules in static priority systems is often less than for dynamic priority systems, we consider this to be of lower importance when there are multiple PCPUs. With the emergence of multi- and many-core processors, it is arguably less important to guarantee the full utilization of every core than it is to provide temporal isolation between threads. In our case, the motivation is to provide temporal isolation between VCPUs supporting one or more threads. It should be noted that additional information such as worst-case execution times are needed to guarantee tasks meet deadlines.

Aside from temporal isolation of VCPUs, one additional factor in the design of Quest is the placement of VCPUs on PCPUs, to reduce microarchitectural resource contention. Guaranteeing a VCPU receives its bandwidth in a specified window of real-time does not guarantee that a thread using that VCPU will make efficient use of the corresponding CPU cycles. For example, a thread may stall on cache misses or memory bus bandwidth contention with other threads co-running on other cores. For this reason, Quest is being developed with a *performance monitoring* subsystem that inspects hardware performance counters to improve VCPU scheduling.

Most modern multicore processors have several performance counters available per core. For example, the Intel Nehalem allows each core to monitor four concurrent performance events, such as the last-level cache misses or hits, while there are eight "uncore" counters to monitor chip-wide events across all cores. We have developed techniques to sample these event counters, to construct estimates of shared on-chip cache usage by individual threads and VCPUs. In this way, we are able to map VCPUs to PCPUs

so that: (1) the VCPUs on any one PCPU have a feasible schedule, and (2) the co-running VCPUs on each PCPU incur the least amount of microarchitectural resource contention. For the latter case, we are developing heuristics to infer cache and memory bus bandwidth contention using events such as cache misses, hits, clock cycle counts and instructions retired. While this is out of scope for this paper, preliminary information on our cache occupancy estimation is available in other work [27].

**I/O VCPUs.** For I/O VCPUs, we have considered several approaches for bandwidth preservation and scheduling. One approach is to use Sporadic Servers, but it is not clear what the most appropriate period should be to satisfy all I/O requests and responses. This is especially problematic when an I/O VCPU is shared amongst multiple tasks that issue I/O requests at different rates. While it is possible to dedicate a separate I/O VCPU to each task issuing I/O requests, so that individual bandwidth and rate requirements can be established, this adds overhead. Instead, it is preferable to use a single I/O VCPU for a given device that is shared across multiple tasks and, hence, Main VCPUs.

We devised a solution in which an I/O VCPU operates as a "Priority Inheritance Bandwidth-preserving Server" (PIBS). With PIBS, each I/O VCPU is specified a certain utilization factor $V_U$, to limit its bandwidth. When an I/O event for this VCPU occurs, the task associated with its occurrence is determined. For example, if some task $\tau$ initiated an I/O request that led to the I/O event being generated, the I/O VCPU inherits the priority of the Main VCPU associated with $\tau$. Since we use rate-monotonic scheduling of VCPUs assigned to a single PCPU, the I/O VCPU inherits a priority inversely proportional to the period of $\tau$'s main VCPU. Moreover, the I/O VCPU assumes a worst-case replenishment period equal to the period of the Main VCPU, $V_{T,main}$. The I/O VCPU budget is limited to $V_{T,main} \cdot V_U$, which is made available at the time of an I/O event. In Quest, all I/O events are processed on behalf of a task associated with a Main VCPU. This includes system tasks and those associated with applications.

When an I/O VCPU is scheduled, the actual budget usage is monitored from the time it starts executing until it terminates. We use a hardware timestamp counter to track budget usage with cycle accuracy. A timeout is set to prevent over-run of the budget. Once an I/O VCPU either times out or completes its task, we update the eligibility time $V_e$ for future invocations of the I/O VCPU by an amount $V_u/V_U$, where $V_u$ is the actual budget used in the most recent invocation.

**Algorithm Descriptions.** We now describe the implementation of the VCPU scheduling algorithms. A Sporadic Server Main VCPU $V$, consists of budget $V_b$, initial capacity $V_C$, period $V_T$, queue of replenishments $V_R$ ordered by time, and current usage $V_u$. I/O VCPUs consist of $V_b$, $V_u$, an eligibility time $V_e$, a utilization limit $V_U$, a single replenishment $V_r$, and a boolean status of "budgeted". $C_{max}$ is defined as $V_T \cdot V_U$ for a given I/O VCPU, where $V_T$ is inherited from the Main VCPU on behalf of which it is serving. A replenishment $r$ is a pair consisting of a time $r_t$ and some amount of budget $r_b$.

The scheduler relies on four VCPU-specific functions: `end-of-timeslice`, `update-budget`, `next-event`, and `unblock`. The first three are used by Algorithm 1. The wakeup routine invokes `unblock`.

**Algorithm 1** `schedule`

**Require:** $V$ is current VCPU.
**Require:** $\bar{V}$ is set of runnable VCPUs.
**Require:** $t_{cur}$ is current time.
**Require:** $t_{prev}$ is previous time of scheduling.
1: Let $\Delta t = t_{cur} - t_{prev}$ and let $T_{prev} = V_T$.
2: Invoke `end-of-timeslice` on $V$ with $\Delta t$.
3: Find $V_{next} \in \bar{V}$ with highest priority and non-zero budget. Invoke `update-budget` on each candidate VCPU before checking its budget.
4: **if** there is no satisfactory $V_{next}$ **then**
5: Enter idle mode and go to step 14.
6: Let $T_{next}$ be the period of $V_{next}$.
7: Select next thread for $V_{next}$.
8: **if** $V_{next}$ has empty runqueue **then**
9: Let $\bar{V}' = \bar{V} \setminus \{V_{next}\}$
10: $V_{next}$ is no longer runnable.
11: **else**
12: Let $\bar{V}' = \bar{V}$.
13: Initially let $\Delta t'$ be equal to the budget of $V_{next}$.
14: **for** each VCPU $v \in \bar{V}$ with higher priority than $V_{next}$ **do**
15: Let $t_e$ be the result of `next-event` on $v$.
16: **if** $t_e$ is valid **and** $t_e - t_{cur} < \Delta t'$ **then**
17: Set $\Delta t' := t_e - t_{cur}$.
18: Set timer to go off after $\Delta t'$ has elapsed.
19: Set $t_{prev} := t_{cur}$ for next time.
20: $V$ is no longer running. $V_{next}$, if valid, is now running.
21: Switch to $V_{next}$ or idle.
**Ensure:** $\bar{V}'$ is now the set of runnable VCPUs.

Algorithm 1 is the entry point into the scheduler architecture. It performs the general work of informing the current VCPU about its usage, finding the next VCPU to run, and arranging the system timer to interrupt when the next important event occurs. The VCPU-specific functionality is delegated to the hooks `end-of-timeslice`, `update-budget`, and `next-event`.

Algorithm 2 enables a task and its associated VCPU to become runnable. This takes care of putting the task and VCPU on their corresponding runqueue, and then invokes any VCPU-specific functionality using `unblock`

and `update-budget`. Finally, the running VCPU is preempted if the newly woken one is higher priority.

**Algorithm 2** `wakeup`

**Require:** Task $\tau$.
**Require:** $CUR$ is currently running VCPU.
**Require:** $V$ is VCPU associated with task $\tau$.
1: Place $\tau$ on runqueue inside $V$.
2: Place $V$ on runqueue.
3: Invoke `unblock` on $V$.
4: Invoke `update-budget` on $V$.
5: **if** $V_b > 0$ **and** ($CUR$ is idle **or** $CUR_T > V_T$) **then**
6: Preempt and invoke scheduler.

The Sporadic Server policy for Main VPUs is based on that described by Stanovich et al [25]. In Algorithm 3 the VCPU has reached the end of a timeslice, so the usage counter is updated and `budget-check` is invoked to update the replenishment queue. If the VCPU has been blocked, then a partially used replenishment may need to be split into two pieces. The $capacity(V)$ formula determines the amount of running time a VCPU can obtain at the current moment. This formula is defined as:

$$head(V) = \min_{r_t}\{r \in V_R\}$$
$$second(V) = \min_{r_t}\{r \in V_R \mid r \neq head(V)\}$$
$$capacity(V) = \begin{cases} 0 & \text{if } head_t(V) > t_{cur} \\ head_b(V) - V_u & \text{otherwise.} \end{cases}$$

$head(V)$ is the earliest replenishment, and $second(V)$ is the one that follows. In Algorithm 4 the capacity of VCPU $V$ is used to set the current value of $V_b$. For a non-running VCPU, the next possible time to run is determined by finding the next replenishment in the queue that has yet to occur, which is expressed in Algorithm 5. When a Main VCPU unblocks, it invokes `unblock-check` to update the earliest replenishment time and perform any replenishment list merges that become possible. Note that when Main VCPUs are created the initial replenishment must be set to the current time for the amount $V_C$ (Algorithm 6).

**Algorithm 3** `MAIN-VCPU-end-of-timeslice`

**Require:** VCPU $V$
**Require:** Time interval consumed $\Delta t$
1: Set $V_u := V_u + \Delta t$.
2: Invoke `budget-check` on $V$.
3: **if** $capacity(V) > 0$ **then** /* Blocked or preempted */
4: **if** $V$ is **not** runnable **then** /* Blocked */
5: Invoke `split-check` on $V$.
6: Set $V_b := capacity(V)$.
7: **else**
8: Set $V_b := 0$.

**Algorithm 4** `MAIN-VCPU-update-budget`

**Require:** VCPU $V$
**Require:** Current time $t_{cur}$
1: Set $V_b := max\{capacity(V), 0\}$.

**Algorithm 5** `MAIN-VCPU-next-event`

**Require:** VCPU $V$
**Require:** Current time $t_{cur}$
1: **return** $\min\{r_t \mid r \in V_R \wedge t_{cur} < r_t\}$ **or** "No event."

**Algorithm 6** `MAIN-VCPU-init`

1: $add(V_R, V_C, t_{cur})$.

The default I/O VCPU algorithm is a Priority Inheritance Bandwidth-Preserving Server (PIBS). It is expressed here as a set of hooks into the VCPU scheduling framework. When an I/O VCPU finishes a timeslice, it updates its budget and usage amounts as shown in Algorithm 7. If it is blocked or out of budget, the eligibility time is advanced and a replenishment is set for that time. Since there is only a single replenishment for an I/O VCPU, the budget can be updated in Algorithm 8 by simply checking the time. For budget management, the only possible event for a non-running I/O VCPU comes from its single replenishment, in Algorithm 9.

**Algorithm 7** `IO-VCPU-end-of-timeslice`

**Require:** I/O VCPU $V$
**Require:** Time interval consumed $\Delta t$
1: Set $V_b := \max\{0, V_b - \Delta t\}$.
2: Set $V_u := V_u + \Delta t$.
3: **if** $V$ is **not** runnable **or** $V_b = 0$ **then**
4: /* Blocked or budget exhausted */
5: Set $V_e := V_e + V_u/V_U$.
6: **if** $V_r$ is unused **then**
7: Set $V_r := r$ where $r_t = V_e$ and $r_b = C_{max}$.
8: **else**
9: Set $V_{r_t} := V_e$.
10: Set $V_u := 0$.
11: Set $V_b := 0$.
12: **if** $V$ is **not** runnable **then** /* Blocked */
13: Set $V$ as **not** "budgeted."

**Algorithm 8** `IO-VCPU-update-budget`

**Require:** I/O VCPU $V$
**Require:** Current time $t_{cur}$
1: **if** $V_r$ is valid **and** $V_{r_t} \leq t_{cur}$ **then**
2: Set $V_b := V_{r_b}$.
3: Invalidate $V_r$.
**Ensure:** $0 \leq V_b \leq C_{max}$

**Algorithm 9** `IO-VCPU-next-event`

**Require:** I/O VCPU $V$
**Require:** Current time $t_{cur}$
1: **if** $V_r$ is valid **and** $t_{cur} < V_{r_t}$ **then**
2: **return** $V_{r_t}$.
3: **else**
4: **return** No event.

**Algorithm 10** `IO-VCPU-unblock`

**Require:** I/O VCPU $V$ associated with I/O task.
**Require:** Main VCPU $M$ waiting for result of I/O task.
**Require:** $t_{cur}$ is current time.
1: **if** $M_T < V_T$ **or not** ($V$ is running **or** $V$ is runnable) **then**
2: Set $V_T := M_T$.
3: **if** $V$ is **not** running **and** $V_e < t_{cur}$ **then**
4: Set $V_e := t_{cur}$. /* I/O VCPU was inactive */
5: **if** $V_r$ is invalid **then**
6: **if** $V$ is **not** "budgeted" **then**
7: Set $V_r$ to replenish for $C_{max}$ budget at time $V_e$.
8: **else**
9: Set $V_{r_b} := C_{max}$.
10: Set $V$ as "budgeted."

**Algorithm 11** `budget-check`

**Require:** VCPU $V$.
1: **if** $capacity(V) \leq 0$ **then**
2: **while** $head_b(V) \leq V_u$ **do**
3: /* Exhaust and reschedule replenishments */
4: Set $V_u := V_u - head_b(V)$.
5: Let $r = head(V)$.
6: $pop(V_R)$.
7: Set $r_t := r_t + V_T$.
8: $add(V_R, r)$.
9: **if** $V_u > 0$ **then** /* $V_u$ is overrun */
10: Set $head_t(V) := head_t(V) + V_u$.
11: **if** $head_t(V) + head_b(V) \geq second_t(V)$ **then**
12: /* Merge into following replenishment */
13: Let $b = head_b(V)$ and $t = head_t(V)$.
14: $pop(V_R)$.
15: Set $head_b(V) := head_b(V) + b$
16: Set $head_t(V) := t$.
17: **if** $capacity(V) = 0$ **then**
18: Set $V$ to background mode.
19: **if** $V$ is runnable **then**
20: $V$ is set to foreground mode at time $head_t(V)$.

When an I/O VCPU unblocks that means a Main VCPU requires some I/O task performed on its behalf. Therefore, the I/O VCPU adjusts its priority according to the period of that Main VCPU. This is performed with a simple comparison of priorities in Algorithm 10. Although this method could result in the I/O VCPU maintaining a higher priority than it deserves over some periods of time, this effect is lessened if jobs are short. A stricter approach would track the precise moment when the I/O VCPU completes service for a specific Main VCPU, and would lower the priority to that of the next highest Main VCPU awaiting service from the I/O VCPU. However, this requires early de-multiplexing in order to figure out when the I/O VCPU has finished processing all I/O for a given Main VCPU, and introduces a loop into the algorithm that is otherwise constant-bounded.

In order to prevent the I/O VCPU eligibility time from falling behind real-time, it is updated to the current time if the I/O VCPU is not running. Then a replenishment is posted for the I/O VCPU of $C_{max}$ budget at the eligibility time. The purpose of the "budgeted" state is to prevent repeated job requests from replenishing the I/O VCPU beyond $C_{max}$ in budget. The "budgeted" state is only reset when the I/O VCPU blocks, therefore the replenishment can only be posted in this way once per eligibility period.

Algorithms 11, 12, and 13 are based on Stanovich et al [25]. These listings include fixes to minor errors we discovered in the original description.

**Algorithm 12** `split-check`

**Require:** VCPU $V$.
**Require:** Current time $t_{cur}$.
1: **if** $V_u > 0$ **and** $head_t(V) \leq t_{cur}$ **then**
2: Let $remnant = head_b(V) - V_u$.
**Ensure:** $capacity(V) = remnant$.
3: **if** $V_R$ is full **then** /* Push remnant into next */
4: $pop(V_R)$.
5: Set $head_b(V) := head_b(V) + remnant$.
6: **else**
7: Set $head_b(V) := remnant$.
**Ensure:** $capacity(V) = capacity(V) - V_u$.
8: $add(V_r, V_u, head_t(V) + V_T)$.
9: Set $V_u := 0$.
**Ensure:** $capacity(V) = capacity(V)$.

**Algorithm 13** `unblock-check`

**Require:** VCPU $V$.
**Require:** Current time $t_{cur}$.
1: **if** $capacity(V) > 0$ **then**
2: Set $V$ to foreground mode.
3: /* Advance earliest replenishment to now. */
4: Set $head_t(V) := t_{cur}$.
5: **while** $|V_R| > 1$ **do**
6: Let $b := head_b(V)$.
7: **if** $second_t(V) \leq t_{cur} + b - V_u$ **then** /* Merge */
8: $pop(V_R)$.
9: Set $head_b(V) := head_b(V) + b$.
10: Set $head_t(V) := t_{cur}$.
11: **else**
12: **return**
13: **else**
14: $V$ is set to foreground mode at time $head_t(V)$.

**Temporal Isolation.** Temporal isolation is guaranteed in our system if the Liu-Layland utilization bound test is satisfied. For a single PCPU with $n$ Main VCPUs and $m$ I/O VCPUs we have the following:

$$\sum_{i=0}^{n-1} \frac{C_i}{T_i} + \sum_{j=0}^{m-1} (2 - U_j) \cdot U_j \leq n \left( \sqrt[n]{2} - 1 \right) \quad (1)$$

Here, $C_i$ and $T_i$ are the budget capacity and period of Main VCPU $V_i$, and $U_j$ is the utilization factor of I/O VCPU $V_j$. A sketch of this proof can be seen by assuming the worst-case scenario of having every I/O VCPU working on behalf of the highest priority Main VCPU $V_h$ in the system. The I/O VCPUs can be treated as budget extensions to Main VCPU $V_h$, which gives this utilization inequality:

$$\sum_{i=0}^{h-1} \frac{C_i}{T_i} + \frac{C_h + \sum_{j=0}^{m-1} (2 - U_j) \cdot U_j \cdot T_h}{T_h} + \sum_{i=h+1}^{n-1} \frac{C_i}{T_i} \leq n \left( \sqrt[n]{2} - 1 \right) \quad (2)$$

When the $U_j$ terms are factored out and simplified this formula leads to Inequality 1.

The term $(2 - U_j) \cdot U_j$ considers the worst-case utilization of an I/O VCPU $V_j$ using the PIBS algorithm. Here, we may see an increase in utilization of $V_j$ by a factor $(2 - U_j)$, as a consequence of the time between eligibility points of the I/O VCPU being dynamic. Specifically, while the utilization between a pair of eligibility points is set to $U_j$, it may be possible that over a period inherited from the corresponding Main VCPU that we have a burst of service exceeding $U_j$. Figure 2 emphasizes this point. Here, $C_{max}/U_j$ is a dynamically-assigned period of $V_j$, based on the corresponding Main VCPU it represents. In this period, an event may be executed at time $e_1$ for $C_{actual} < C_{max}$, leading to an updated eligibility point $e_2$ when $V_j$ can execute again. For example, suppose $V_j$ operates on behalf of a Main VCPU $V_i$ with period $T_i = 4$, and $U_j = 0.5$. If $C_{actual} = 1$ and $e_1 = 0$, then $e_2$ is set to 2. Then, suppose the I/O VCPU executes for $C_{max} = 2$. Even though the next eligibility point (not shown) is set to 6, the total utilization over the window $C_{max}/U_j$ is $3/4$. This exceeds $U_j$.

From Figure 2, the worst-case utilization of $V_j$ over the interval $C_{max}/U_j$ is:

$$\frac{(C_{max}/U_j - C_{max}) \cdot U_j + C_{max}}{C_{max}/U_j} = (2 - U_j) \cdot U_j \quad (3)$$

This accounts for the total overheads of each I/O VCPU in Inequality 1. Note that in practice, Quest associates a separate scheduling queue per physical CPU (PCPU). Scheduling analysis such as that described above can be applied to each PCPU separately. Global scheduling is out of the scope of this paper, although we are considering migration techniques to relocate VCPUs and threads on different PCPUs at runtime.

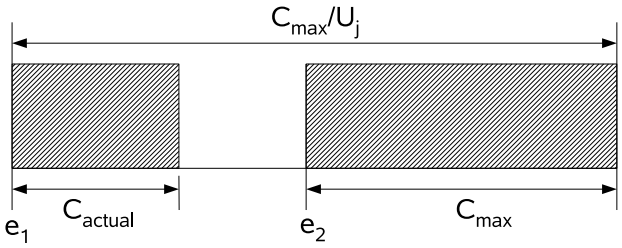


**Figure 2. Worst-case I/O VCPU Utilization**

## 3 Experimental Evaluation

We conducted a series of experiments on a single core of an Intel Core2 Extreme QX6700 running at 2.66 GHz with 4GB of DDR2 SDRAM. In what follows, we used a network interface card based on a Gigabit Ethernet device from the Intel 8254x (a.k.a. "e1000") family that connects via the PCI bus. A UHCI-based USB host controller is used in a series of tests involving reading from a Mass Storage solid state disk of 1GB total size. A CD-ROM drive is also connected via Parallel ATA and sectors are read using the PIO method. We developed drivers for USB, CD-ROM and the e1000 NIC from scratch, and although they are not optimized for efficiency at this stage they serve as examples to show how our system responds to I/O events. The focus of our experiments is on the temporal isolation and scheduling overheads of our Quest system. Although Quest is being developed as an SMP system we do not consider the mapping of VCPUs to PCPUs in this paper.

All bandwidth and CPU usage measurements have been performed with an average over a 5-second window. We use the processor timestamp counter to track elapsed clock cycles using the `rdtsc` machine instruction. We verified that the timestamp counter increments at the same rate as unhalted clock cycles on our machine. Power management features were disabled.

Performance data is cached in memory and then is reported through the serial port by a logging thread that runs under the same conditions as any other thread in the system. The logging thread continually attempts to fetch characters from a ring buffer in shared memory, or busy-waits if there is nothing to read. In addition, any number of CPU-bound threads can be generated for testing purposes. These threads run in user-space and simply increment a counter, occasionally printing a character on the screen. The CD-ROM and USB test threads both use filesystem facilities to read 64kB

of data from a file repeatedly on the corresponding device.

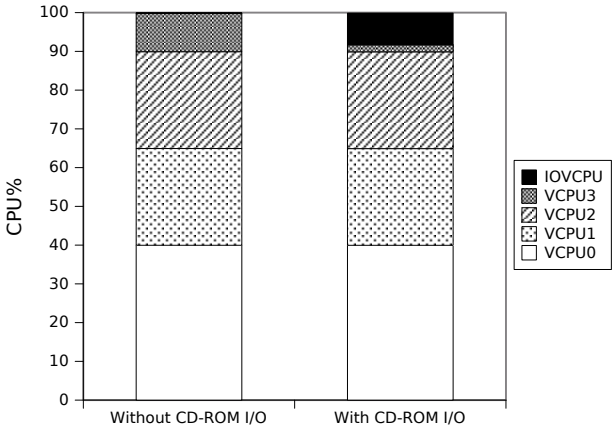


**Figure 3. Effect of CD-ROM I/O on VCPUs**

| VCPU | $V_C$ | $V_T$ | threads |
|---|---|---|---|
| VCPU0 | 2 | 5 | CPU-bound |
| VCPU1 | 2 | 8 | Reading CD, CPU-bound |
| VCPU2 | 1 | 4 | CPU-bound |
| VCPU3 | 1 | 10 | Logging, CPU-bound |
| IOVCPU | 10% | | ATA |

**Table 1. Effect of CD-ROM I/O on VCPUs**

In Figure 3 the VCPUs are programmed with the settings[3] of Table 1. The first run is conducted with the CD-reading thread disabled. The first column in the graph shows the CPU usage of each VCPU in the system when there is no CD-ROM I/O. For the second run, the CD-reading thread is enabled. This thread runs on VCPU1 which has lower priority than all VCPUs except VCPU3. The second column shows that as a result, only VCPU3 has been forced to sacrifice its utilization, while the higher priority VCPUs remain isolated.

The I/O VCPU algorithm is based on the notion of a Priority Inheritance Bandwidth-preserving Server (PIBS), as opposed to the Main VCPU algorithm which is a Sporadic Server (SS). A scenario is described in Table 2 which is first run with a Priority Inheritance Bandwidth-preserving Server I/O VCPU and then is run with a Sporadic Server I/O VCPU. In both cases, at approximately time $t = 50$, the network interface begins receiving packets from an ICMP ping-flood. We used ping-flooding rather than a bulk data transfer because the short intervals between packet arrivals stress the scheduling capabilities of Quest more than would be the case with larger packets at longer intervals.

Figure 4 shows how the Sporadic Server I/O VCPU has a greater and more variable amount of scheduler overhead

[3]The base unit for values of $C, T$ is $100\mu$sec unless otherwise specified.

| VCPU | $V_C$ | $V_T$ | threads |
|---|---|---|---|
| VCPU0 | 1 | 20 | CPU-bound |
| VCPU1 | 1 | 30 | CPU-bound |
| VCPU2 | 10 | 100 | Network, CPU-bound |
| VCPU3 | 20 | 100 | Logging, CPU-bound |
| IOVCPU | 1% | | Network |

**Table 2. PIBS vs SS Scenario**

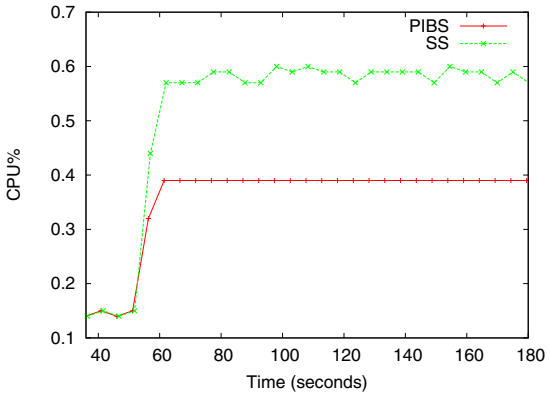


**Figure 4. Scheduler Overhead**

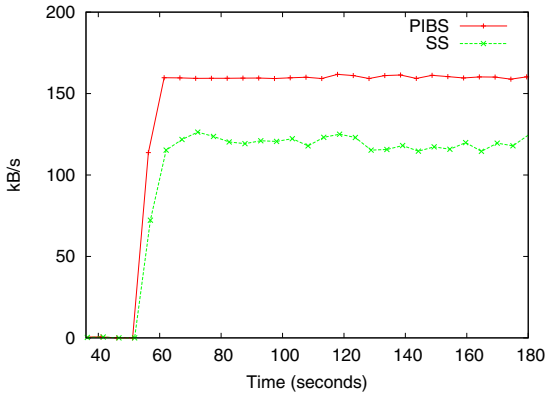


**Figure 5. Network Bandwidth**

than the PIBS I/O VCPU. Figure 5 compares the network bandwidth of the two experiments, showing that the PIBS I/O VCPU achieves a significantly greater throughput.

The primary advantage of the PIBS I/O VCPU is the simple and regular provision of budget over time. On the other hand, the SS I/O VCPU obeys the rules regarding splitting and merging of replenishments, which causes a larger amount of overhead. Since the network task only runs for short bursts before blocking, the replenishments split into fragments, and the queue is quickly filled up. In this case the maximum replenishment queue length was set to 32. We arbitrarily chose 32 to be the maximum number of replenishments per sporadic server. The actual choice of

maximum queue length is application-specific, but frequent blocking operations can nonetheless result in a completely filled queue for situations such as when there is high I/O activity. When the queue reaches its maximum length, the Sporadic Server algorithm discards effective budget in order to compensate, therefore the SS I/O VCPU loses bandwidth.

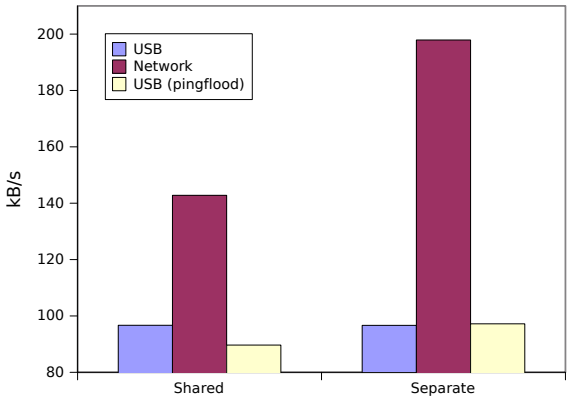


**Figure 6. Shared vs Separate I/O VCPUs**

| VCPU | $V_C$ | $V_T$ | threads |
|---|---|---|---|
| VCPU0 | 30 | 100 | USB, CPU-bound |
| VCPU1 | 10 | 110 | CPU-bound |
| VCPU2 | 10 | 90 | Network, CPU-bound |
| VCPU3 | 100 | 200 | Logging, CPU-bound |
| IOVCPU | 1% | | USB, Network |

| VCPU | $V_C$ | $V_T$ | threads |
|---|---|---|---|
| VCPU0 | 30 | 100 | USB, CPU-bound |
| VCPU1 | 10 | 110 | CPU-bound |
| VCPU2 | 10 | 90 | Network, CPU-bound |
| VCPU3 | 100 | 200 | Logging, CPU-bound |
| IOVCPU | 1% | | USB |
| IOVCPU | 1% | | Network |

**Table 3. Shared vs Separate I/O VCPUs**

Figure 6 is based on the settings specified in Table 3. The first experiment is conducted with a single I/O VCPU limited to 1% utilization for use by both USB and network traffic. The second experiment separates these sources of I/O onto different I/O VCPUs, each with their own 1% utilization. In both cases, the bandwidth of data read from USB was sampled first when there were no incoming network packets, and then sampled during a ping-flood. The first set of bars in the graph show that when USB and network traffic share an I/O VCPU, the USB bandwidth and network bandwidth are both degraded. The second set of bars shows that separate I/O VCPUs isolate the corresponding I/O devices from each other.

The reason why the I/O VCPU budget for the USB driver

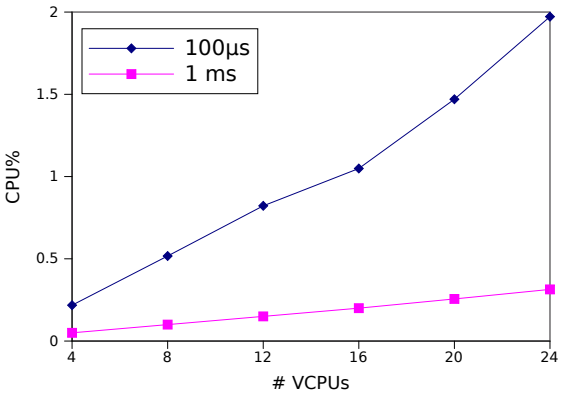


**Figure 7. Scheduler Overhead**

is 1% in both cases is to ensure that measured USB bandwidth is equal in both experiments, prior to the ping-flood. Observe that this experiment is not about raw bandwidth but how device bandwidth can be affected when multiple devices share the same I/O VCPU. In contrast, when they have separate I/O VCPUs they achieve temporal isolation.

As the number of VCPUs in the system increases, the overhead of running the scheduler algorithm also increases. In Figure 7 there are two sets of experiments that show this increase is basically linear. The experiments are conducted by creating different numbers of VCPUs, each with one CPU-bound thread. The scheduling overhead is the percentage of cycles spent in the scheduler during a 5 second window. These experiments were run with a basic budget unit of $100\mu$s, and were repeated with a basic unit of 1ms. The scheduling overhead is higher overall in the former case since the scheduler is invoked more frequently in any given window of 5 seconds.

## 4 Related Work

While many real-time operating systems exist today, it is less common to see systems with explicit support for temporal isolation using resource reservations or budgets. One such system that is built around the notion of resource reserves is Linux/RK[15]. The concept of a reserve in Linux/RK was derived and generalized from processor capacity reserves [13] in RT-Mach. In more recent times there has been similar work on resource containers to account for resource usage [3]. Each time-multiplexed reserve in the system has a budget $C$, interval $T$, and deadline $D$, assigned to it so that utilization can be specified. Linux/RK requires apriori knowledge of application resource demands and relies on an admission control policy to guarantee a reasonable global reserve allocation. In contrast, Quest focuses on the temporal isolation between tasks and system events

using a hierarchy [21, 18] of virtual servers, acting as either Main or I/O VCPUs.

Redline [28] is a system that focuses on predictability for interactive and multimedia applications. It also has the notion of budgets and replenishments, but the task scheduling model appears similar to that used in Deferrable Servers [26, 5]. Given Redline's focus, the system differentiates interactive and best-effort tasks, and optimistically accepts new tasks based on the actual usage of the system. In the presence of overload, a load monitor will select an interactive victim and downgrade it to best-effort in order to fulfill response time requirements of other interactive tasks. Quest shares some of Redline's properties, but the focus is on dependent scheduling of tasks and system events, especially events triggered in response to I/O requests. In contrast to both Redline and Linux/RK, Quest allows I/O events to be processed at priorities inherited from virtual servers responsible for executing tasks, for whom I/O event processing is being performed.

The HARTIK kernel [1] supports the co-existence of both soft and hard real-time tasks. To ensure temporal isolation between hard and soft real-time tasks, the soft real-time tasks are serviced using a Constant Bandwidth Server (CBS). A CBS has a current budget, $c_s$ and a bandwidth limited by the ratio $Q_s/T_s$, where $Q_s$ is the maximum server budget available in the period $T_s$. When a server depletes all its budget it is recharged to its maximum value. A corresponding server deadline is updated by a function of $T_s$, depending on the state of the server when a new job arrives for service, or when the current budget expires. CBS guarantees a total utilization factor no greater than $Q_s/T_s$, even in overloads, by specifying a maximum budget in a designated window of time. This contrasts with work on the Constant Utilization Server (CUS) [6] and Total Bandwidth Server (TBS) [24], which ensure bandwidth limits only when actual job execution times are no more than specified worst-case values. CBS has bandwidth preservation properties similar to that of the Dynamic Sporadic Server (DSS) [7] but with better responsiveness.

CBS, CUS, TBS and DSS all assume the existence of server deadlines. We chose not to assume the existence of deadlines for VCPUs in Quest, instead restricting VCPUs to fixed priorities. This avoided the added complexity of managing dynamic priorities of VCPUs as their deadlines change. Additionally, for cases when there are multiple tasks sharing a fixed-priority VCPU, the execution of one task will not change the importance of the VCPU for the other tasks. That said, our priority inheritance policy for I/O VCPU scheduling (PIBS) has some similarities to CUS and TBS. It assigns priorities to I/O VCPUs based on the priority of the task (specifically, its Main VCPU) associated with the I/O event to be serviced. Eligibility times are then set in a manner similar to how deadlines are updated with CUS and TBS, except we use eligibility times to denote when the I/O VCPU can resume usage of processor cycles without exceeding its bandwidth capacity. Observe that with PIBS, the next server eligibility time is not set until all the I/O VCPU budget is consumed, or an I/O event completes within the allowed budget. In comparison, CUS and TBS determine deadlines *before* execution assuming knowledge of WCET values.

Motivation for both Main and I/O VCPUs in Quest was provided by our earlier work to integrate the scheduling of interrupts and tasks [29]. While others have proposed methods to unify task and interrupt scheduling [10], or have considered bandwidth constraints on device driver execution [9], Quest attempts to combine both the prioritization of I/O events and budget limits for their handling with task scheduling. In doing so, we describe a method to integrate asynchronous event processing for both device interrupts and tasks waking up after the completion of blocking (e.g., I/O) operations.

## 5 Conclusions and Future Work

This paper describes the scheduling infrastructure in the Quest operating system, based around the concept of a VCPU. Temporal isolation between VCPUs is achieved using variants of well-known bandwidth preservation policies. Quest schedules VCPUs on physical processors, while software threads are scheduled on VCPUs. Using Main and I/O VCPUs, we are able to separate the CPU bandwidth consumed by tasks from that used for I/O processing. Although Quest is flexible enough to support different scheduling policies, we describe an approach that uses Sporadic Servers for Main VCPUs, and a bandwidth preserving technique with priority inheritance (PIBS) for I/O VCPUs.

We show how PIBS has lower scheduling overhead than a Sporadic Server and also how it achieves higher throughput, by avoiding the costs of managing budget replenishment lists. Our Sporadic Server approach is based on that described by Stanovich et al [25], which corrects for defects in the POSIX specification. This type of server supports the fragmentation of resource budget over time. In situations where servers are implemented with finite budget replenishment lists, it is possible to fill these lists as a consequence of many short bursts of I/O processing. We have observed situations where budget lists having up to 32 members quickly fill to capacity due to numerous network interrupts, which affects throughput. However, in situations where tasks execute for relatively longer bursts of time than those needed for I/O events, Sporadic Servers seem more suitable. For a system in which budget allocations are made in units of 1ms we have observed that a Quest system with 24 Main VCPUs

incurs about $0.3\%$ physical CPU overhead. We believe it is possible to build a highly-scalable system using VCPUs as the base entities for scheduling. In future work, we will investigate Quest's performance when using more than one core of a multicore processor. In particular, we aim to show the effectiveness of hardware performance monitoring for co-runner selection of VCPUs to avoid effects such as cache conflict misses and memory bus bandwidth contention.

NB: The Quest source code is available upon request.

## Acknowledgment

The authors would like to thank the anonymous reviewers for their feedback that has helped in the writing of this paper.

## References

[1] L. Abeni, G. Buttazzo, S. Superiore, and S. Anna. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-time Systems Symposium*, pages 4–13, 1998.

[2] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: 12th Intl. Conf. on Architectural Support for Prog. Langs. and Operating Systems*, pages 2–13, 2006.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[5] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *IEEE Real-Time Systems Symposium*, pages 68–78, 1999.

[6] Z. Deng, J. W. S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment, 1997.

[7] T. Ghazalie and T. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, (9), 1995.

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

[9] M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan, and A. Wang. Modeling device driver effects in real-time schedulability analysis: Study of a network driver. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, 2007.

[10] 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*, April 2006.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[12] lwIP: http://savannah.nongnu.org/projects/lwip/.

[13] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: An abstraction for managing processor usage. In *Proc. 4th Workshop on Workstation Operating Systems*, pages 129–134, 1993.

[14] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[15] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in Linux. In *Proc. 19th IEEE Real-Time Systems Symposium*, 1998.

[16] G. Parmer and R. West. Mutable Protection Domains: Towards a component-based system for dependable and predictable computing. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, December 2007.

[17] G. Parmer and R. West. Predictable interrupt management and scheduling in the Composite component-based system. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, Barcelona, Spain, December 2008.

[18] J. Regehr. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 3–14, 2001.

[19] RTAI: https://www.rtai.org/.

[20] Real-Time Linux: http://www.fsmlabs.com.

[21] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *RTSS*, pages 2–13, 2003.

[22] 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.

[23] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems Journal*, 1(1):27–60, 1989.

[24] M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *IEEE Real-Time Systems Symposium*, December 1994.

[25] M. Stanovich, T. P. Baker, A.-I. Wang, and M. G. Harbour. Defects of the POSIX sporadic server and how to correct them. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010.

[26] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environment. *IEEE Transactions on Computers*, 44(1):73–91, January 1995.

[27] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online cache modeling for commodity multicore processors. In *19th International Conference on Parallel Architectures and Compilation Techniques (PACT), regular poster*, September 2010. See also Boston University TR 2010-015 or VMWare-TR-2010-002, July 2010.

[28] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Redline: First class support for interactivity in commodity operating systems. In *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[29] Y. Zhang and R. West. Process-aware interrupt scheduling and accounting. In *Proceedings of the 27th IEEE Real Time Systems Symposium*, December 2006.