Hijack: Taking Control of COTS Systems for Real-Time User-Level Services *

Gabriel Parmer and Richard West

Computer Science Department
Boston University
Boston, MA 02215
{gabep1,richwest}@cs.bu.edu

Abstract

This paper focuses on a technique to empower commercial-off-the-shelf (COTS) systems with an execution environment, and corresponding services, to support real-time and embedded applications. By leveraging COTS systems, we are able to reduce the potentially expensive maintenance and development costs of proprietary solutions. We describe a system called "Hijack" that enables user-level services to take control of features such as CPU scheduling, interrupt handling and synchronization. In contrast to other approaches that support real-time tasks within the kernel of commodity systems such as Linux, Hijack provides the basis for predictable thread execution at user-level. No changes to the kernel source code are required to support this approach. Instead, Hijack works by using a combination of kernel module support and an interposed execution environment between traditional process address spaces and the kernel. This technique enables system calls and hardware interrupts to be intercepted with bounded latencies via the kernel module, that passes control to a user-level real-time executive. From within the executive, system-wide services and policies can be deployed to over-ride certain features of the underlying kernel, while still leveraging base kernel services where appropriate.

Using this technique, we show how a vanilla Linux system can be hijacked to support predictable service execution using a series of user-defined policies. In particular, we show how to deliver and process asynchronous events with bounded latency, using interposition agents within a Hijack execution environment. Results show that for real-time streaming applications, Hijack is able to receive and process packets with significantly lower loss rates and jitter compared to using alternative application-level processes for the same task.

*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

In order to eliminate the costs of proprietary systems and special purpose hardware, many real-time and embedded computing platforms are being built on commodity operating systems and generic hardware. For example, RTLinux [17], RTAI [16] and the Resource Kernel [11] are real-time systems built using Linux, with modifications to ensure predictability. Commodity systems are a desirable basis for real-time computing platforms because they potentially reduce development and maintenance costs compared to special purpose systems. They also provide a common code-base, including numerous device drivers, which can potentially be utilized in a diverse range of computing environments.

While real-time systems such as those mentioned above leverage Linux, they either require real-time tasks to run within the kernel protection domain (which is the case with RTLinux Free and RTAI) or they require changes to the kernel source code (e.g., to add resource reservation policies, deferrable interrupt mechanisms and high resolution timing [19]). The goal of our work is to empower commodity systems such as Linux with safe, predictable and efficient execution environments, that isolate real-time tasks and application-specific services from the kernel. In achieving this goal, the aim is to avoid changes to the underlying kernel source code to establish a user-space environment for application-specific services.

In our previous work on user-level sandboxing (ULS) [23], we showed how to establish a predictable execution environment at user-level using commonly-available hardware protection mechanisms (such as paging). Our ULS scheme enables real-time tasks and custom services to be deployed and then executed with bounded and low latency when corresponding system events occur. However, while ULS provides the basis to implement "first-class services" at user-level [1], it does not provide a complete ex-

[1] We define a "first-class service" at user-level as having the same privileges and capabilities, where possible, as kernel services, with the exception that the kernel can revoke access rights to such services if they abuse

ecution environment necessary to over-ride all inherently non-real-time services of the host kernel. Likewise, ULS still requires application-specific services that rely on kernel services to use the standard system call interface, which is often insufficient to capture the requirements of real-time applications.

This work, therefore, is motivated by the desire to take the philosophy of user-level sandboxing, to build a real-time execution environment on top of a non-real-time commodity operating system. The resultant "Hijack" system uses a kernel loadable module to intercept system calls and hardware interrupts, so they may be directed to a user-space execution environment which enforces predictable service execution. Henceforth, we will refer to Hijack's user-space execution environment as a "real-time executive", which is essentially a protection domain (similar to that in ULS) mapped into a specific virtual address range in all real-time processes. The purpose of this real-time executive is to manage interposition code that mediates access to system resources in a controlled and predictable manner. Fundamentally, then, Hijack offers the capabilities to interpose user-level code between both system calls and hardware interrupts for predictable real-time execution, whereas our ULS work focuses on low-latency execution of application-specific services without recourse to the underlying kernel scheduler (that is inherently non-real-time).

By interposing code at the system interface we are able to modify the behavior of existing system calls, or add new ones, thereby potentially bridging the semantic gap between application needs and the service provisions of the underlying kernel. For example, we could have system calls that set priorities and timeslices be intercepted and handled in a real-time executive using a completely different scheduling policy to that of the underlying kernel. As will be seen in later sections, Hijack offers the ability to redefine and add application-specific services that can be built on pre-existing kernel functionality. The degree to which the underlying kernel supports predictability ultimately dictates the predictability of Hijack services that rely on kernel functions. For this reason, we focus on soft real-time support (e.g., for multimedia streaming and resource partitioning for virtual machines) due to the unpredictability of low-level services in COTS systems such as Linux. However, the Hijack philosophy is largely independent of the underlying kernel and, should that kernel offer highly predictable and bounded service execution (e.g., in RTLinux) there is no reason why Hijack services could not be tailored to hard real-time applications also.

The next section describes the implementation of Hijack in more detail, as it relates to an underlying Linux kernel. This is followed in Section 3 by an evaluation of Hijack using a series of experiments to show the predictability and efficiency of services deployed in a user-space real-time executive. Related work is then outlined in Section 4. Finally, conclusions and future work are discussed in Section 5.

their privileges.

2. Hijack: Implementation Details

Figure 1 shows the Hijack system architecture.

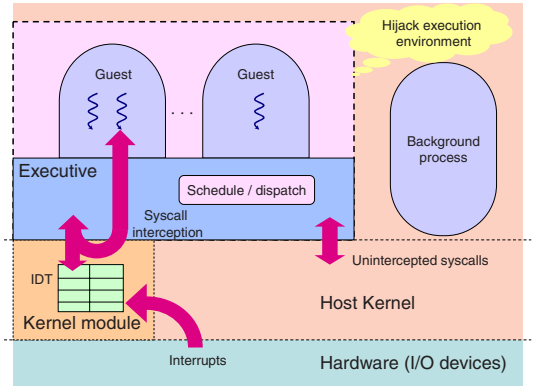

Figure 1. The Hijack system architecture.

2.1. The Hijack Execution Environment

The Hijack execution environment consists of a *real-time executive*, and a number of "guest" real-time tasks. The system architecture allows for the non-real-time background processes to co-exist and interact with the host kernel if so desired. However, for the purposes of enforcing predictable execution of real-time tasks and services, Hijack requires changes to the page table mappings of all guest address spaces so they have a traditional process-private area and a shared region for the executive.

Hijack's execution environment is similar to that in our prior work on ULS [23], but leverages page global bits rather than superpages to minimize the cost of invoking and returning from executive services. Hijack relies only on paging, which is more commonly found on modern processors. With Hijack, all processes' page table entries are modified so that, on the x86 architecture, their global bits are set by default. A shared virtual address space is reserved in each guest for one or more pages of the real-time executive. When a thread is executing within the process-private address space of a guest, the executive's address memory area is not mapped into the guest. This guarantees that application-specific real-time services loaded into the executive are not accessible to guest code executing *outside* the executive, thereby enforcing a degree of memory protection. In contrast, when the Hijack real-time executive is activated, it is mapped into the address space of the current guest using traditional memory mapping techniques.

Observe that in our ULS approach, a sandbox area is mapped a-priori into every process address space and marked as being kernel-accessible until it is activated (at which time the page, or pages, of the sandbox are set to be user-level accessible). The advantage of the Hijack approach is that the executive can support code that uses memory-mapping, while this is not possible using ULS. In fact, we explicitly disable the `mmap()` functionality within the standard C library in our ULS work. Whereas we take advantage of $4MB$ super-pages on the Pentium-class processors in ULS, here we instead rely on global page bits.

On the Pentium-class processors the global bit within a page table entry controls whether or not that page is flushed from the translation look-aside buffer (or, TLB, used to cache virtual-to-physical address translations) when there is a context-switch. By context-switch, we mean a change to the page tables used to identify the address space of a given protection domain. On the Pentium processor, the `CR3` register acts as a page directory base register and stores the physical address of a two-level page-table hierarchy for the current execution context. Any modifications to the value of `CR3` forces all non-global pages to be flushed from the TLB.

By careful manipulation of page global bits, control flow from Hijack's real-time executive back to a process-private guest area results in only the page table mappings of the executive being flushed from the TLB. In this manner, code executing within a guest's private memory area cannot inadvertently access addresses *within* the executive, while also ensuring the TLB does not have to be reloaded for the guest. Hence, this leads to a relatively fast method for safely switching between the real-time executive and guest memory areas. It is worth noting that one could individually invalidate all pages of a Hijack executive rather than flushing all non-global pages (by modifying `CR3`) but experience shows this to be more expensive for moderately-sized executives occupying at least several page frames. Admittedly, switching between Hijack guest processes, or one guest process and a background non-real-time process (operating outside Hijack's control) leads to increased cost due to having to toggle page global entries in the `CR4` register of the processor. However, one of the main thrusts of this work is to optimize for communication and protection between real-time guests and the Hijack executive. As seen in the microbenchmarks in Section 3.1 these costs are not prohibitive. If the increased costs for page-table switching are deemed unacceptable for a given application, we can easily modify our approach to use super-pages as in our ULS work. In fact, we can even leverage hardware-specific features such as segmentation, to implement a small-space [20] execution domain but the focus of this work is to provide an environment in which system policies can be redefined or introduced for application-specific needs.

A guest executing within the Hijack execution environment behaves essentially like any traditional process running on the host system. The only difference is that guests execute only when scheduled by the Hijack executive. In turn, the executive is activated by upcalls from a Hijack kernel module that intercepts interrupts generated either by hardware (e.g., devices or the CPU itself) or software (e.g., due to faults or system calls). In this manner, the timer interrupt can be bound to an upcall into the executive to update various state variables pertaining to the execution time of guests, and also to schedule new guest threads when appropriate. In the current approach, however, the Hijack executive has a single thread of control, which can be thought of as a virtual processor for all subsequent guest threads. The executive thread of control, which invokes the Hijack scheduler and dispatcher is itself scheduled by the host kernel. To ensure the predictable execution of the scheduler within the executive, its corresponding thread is assigned the highest priority amongst all threads seen by the host kernel.

Observe that for all types of kernel events, including I/O completion and timer expiration, predictable execution of code within the executive is ensured because it is the highest priority thread in the system. Given this scenario, any scheduling policy can be hierarchically composed within the executive, as is the case with the work on Hierarchical Loadable Schedulers (HLS) [15]. Additionally, the Hijack execution environment has the capability to manage and multiplex between multiple guest address spaces. This capability is provided by the kernel module that creates and switches between address spaces on behalf of the executive.

Given that we wish to utilize the services of the host system, guests are allowed to link with libraries and make system calls. Any system call from a guest is automatically redirected to the Hijack kernel module. For our current implementation using Linux 2.6.13 on the x86 architecture, system calls are made using the `SYSENTER` instruction. We are able to redirect control flow via system calls into our kernel module by carefully setting the appropriate model specific registers (MSRs) associated with `SYSENTER` to indirectly update the `CS`, `EIP`, `SS` and `ESP` registers. These affect the stack and instruction pointer (i.e., program counter) values when system calls are made. For Linux systems that use the old `INT 0x80` software interrupt to perform system calls, the Hijack kernel module modifies the interrupt descriptor table (IDT) to remap vector entry `0x80` to an address from where it can execute system call interception code. This same method of modifying the IDT is used to intercept other interrupts, and exceptions such as page faults, so that the Hijack executive can execute its own handler code via upcalls from the kernel module.

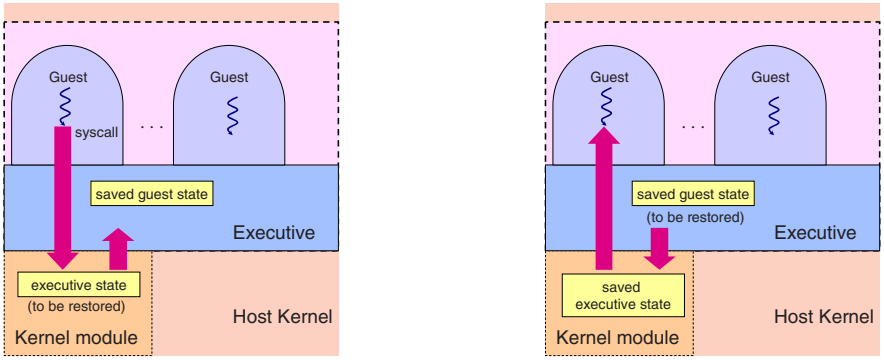

Figure 2. (a) System call interposition, and (b) return from the executive to a guest.

2.2. The Hijack Interposition Mechanism

The Hijack application programming interface allows for interposition agents [8, 5] to be loaded and invoked as part of the execution path of system calls and interrupt handlers. For example, a system call can be intercepted by interposition code that changes the values of its arguments, replaces the system call, and/or enshrouds the system call with added functionality. In this way, the Hijack executive can perform validity checks on guest service requests to ensure quality of service (QoS) guarantees are met without jeopardizing the predictability of other guests in the system. Alternatively, the semantic gap imposed by an application-agnostic system interface of, say, UNIX can be bridged with functionality in the executive, to ensure the needs of applications are better met by the underlying system services. For example, the Hijack executive may include interposition code for POSIX system calls such as `sched_setscheduler()` and `sched_setparam()` on a vanilla Linux system, so that a policy of type `SCHED_OTHER` mimics that of earliest-deadline first scheduling.

In this paper, we focus on Hijack's ability to support system call interposition and controlled guest execution for the purposes of soft real-time execution on non-real-time COTS systems. To support this, Hijack has to manage both the flow of control originating with a system call in a guest that leads to execution in the executive, and also the return of control from the executive back to the corresponding guest. When a guest issues a system call (Figure 2(a)), the Hijack executive is unmapped and therefore inaccessible in the guest address space. However, at the time of the trap to the host kernel, the Hijack kernel module intercepts the system call and maps the page (or pages) of the executive into the calling guest address space. This is done by updating appropriate entries in the page directory (`PGD`) of the current execution context. At this point, the machine (register) state is stored by the kernel module on behalf of the current guest, $g_{current}$. Observe that a guest's state is stored in a well-defined location *within* the executive, so that the executive is able to modify appropriate state values as necessary. The kernel module then disables subsequent interception of system calls since these will be made within the executive as part of any interposition code associated with the guest's system call. Finally, control must be passed to the executive so that specific interposition code can execute. This is done by reloading machine state for a specific entry point into the executive.

After the interposition code finishes, control flow needs to return to either $g_{current}$ or perhaps another guest thread as dictated by the scheduling policy within the executive (Figure 2(b)). First, the return from the interposition code traps into the kernel module where the machine state on behalf of the executive can be stored. The previously saved state of the guest chosen by the executive to resume execution is now loaded into the processor registers. If the executive switches between guests, it also changes the active page tables. Regardless, access to the pages of the executive are then disabled to ensure guest code cannot affect its integrity. This is done by clearing the `PGD` entries for the executive in the context of the selected guest, and also flushing the TLB to remove any stale mappings. System call interposition is re-enabled so that subsequent system calls within the guest pass through the executive.

Page fault interception. Observe that the Hijack system architecture includes page fault interception, in additional to system call interception. This is necessary because the host system (here Linux) maintains software mappings for the executive in the current address space, and we should not allow a guest to access the executive directly from user-space. This would potentially allow a guest to violate the integrity of the executive. Observe that a guest attempting to access a virtual address within the executive, when the executive is unmapped, will lead to a page fault. Ordinarily, the host kernel will map the missing page into the caller's address space but, with Hijack, we simply disallow this by interposing code on the page fault hardware interrupt. In fact, Hijack's interposition mechanism has the capability to take some form of action when a potential safety violation

like this occurs, such as aborting the offending guest.

In other faulting situations, such as a guest causing a segmentation violation, the host kernel can deliver a `SIGSEGV` signal to the executive. This way, the executive can take appropriate action rather than having itself being terminated by a guest. We will discuss event notification issues further in the following subsection.

2.3. Kernel Event Notification

The Hijack system architecture allows user-defined executive code to execute in response to various system events, in particular those triggered via signals. In essence, asynchronous event notification can be handled with bounded and lower latency than would be possible if we had to switch to a specific guest to handle a given event, or can be vectored to a specific guest at the executive's discretion.

If a signal occurs during the execution of a guest (rather than executive code) it is delivered to a function called `signal_handler()`, located $8KB$ below the executive's base virtual address (see Figure 3). Using an alternative stack whose space is allocated above the executive memory region, the `signal_handler()` function issues a special system call notification down into the Hijack kernel module. This system call is intercepted as explained in Section 2.2, so that the executive is mapped into the guest's virtual address space. Control flow passes to the executive which is notified of the signal generated during the guest's execution so that it can now be handled in an executive-defined manner. Before the specific signal is handled, the executive copies guest state saved by the kernel when the signal occurred to the area where guest's state is stored (so that the executive can later re-enable it), and then the executive re-enables signals.

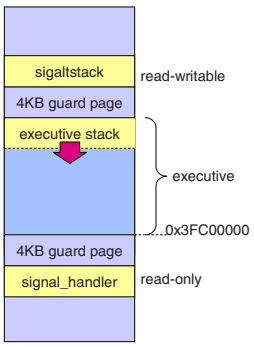

Figure 3. Hijack memory layout.

If a signal occurs during execution within the executive, control flow passes to the `signal_handler()` function, which can detect that the executive's memory region is accessible. This function directly invokes the signal handling routine defined within the executive. When complete, the signal handler executes a `sigreturn` system call to return to the state of the executive's execution context before the signal event.

It would be straightforward to modify the kernel's source code to directly deliver a signal to the executive. However, because we want to avoid kernel source code changes to support Hijack, we use the method described above to differentiate the occurrence of a signal during guest versus executive execution.

3. Experimental Evaluation

All experiments are performed on IBM xSeries 305 e-server machines with Pentium IV, 2.4 GHz processors and 904 MB of available RAM. Each computer has a tigon3 gigabit Ethernet card, connected by a switched gigabit network. We use Linux version 2.6.13 as the host operating system with a clock-tick (or *jiffy*) set to 10 milliseconds.

3.1 Microbenchmarks

Fundamentally, for the Hijack system to provide execution environments that are both efficient and predictable for the guests executing in them, the interactions between the guests and the executive, and between the executive and the host kernel must not impose an unreasonable overhead. Thus to investigate the viability of the approach, Table 1 presents microbenchmarks pertinent to the performance of the system. Additionally, the performance of standard UNIX mechanisms is included for comparison. Each microbenchmark was run 10000 times and the average reported.

Operation	Cost in CPU cycles
System Call	430
RPC from Guest to Executive & back to Guest	2900
Trap from Guest to Executive	1250
Interposition using POSIX `ptrace`	26000
RPC Between Two Hijack Guests (Separate Page Tables)	9250
RPC Between Two Processes Using UNIX Pipes	14500

Table 1. Hijack system performance.

Protected interposition is fundamental to Hijack and allows the executive to translate guest requests for service into predictable and real-time aware forms, while the executive remains isolated from the guests. The efficiency of interposition is examined in the first half of the table. In this case,

when a guest requests service by making a system call, Hijack traps to the executive, which makes that system call on the guest's behalf, then returns to the guest. The total cost of interposition, therefore, includes both the cost of a system call (430 cycles) and the round-trip cost of RPC between the guest and executive (2900 cycles). Cumulatively, this is nearly eight times more efficient than performing interposition with the POSIX `ptrace` mechanism. The `ptrace` microbenchmark includes the cost of retrieving the guest register state to identify the requested system call. We acknowledge that `ptrace` is intended as a debugging interface and is therefore not focused on efficiency, let alone predictability, but it is essentially the only facility on UNIX systems to support interposition. In fact, systems such as User-Mode Linux [21] provide a virtual execution environment for guest Linux processes using `ptrace`, in the absence of host kernel modifications. Note that from Table 1, it takes approximately 1650 cycles to return from the executive to the guest. This is the round-trip cost between the guest and executive minus the cost of a one-way trap to the executive. Returning from the executive to the guest is more expensive than the contrary direction, as all the executive's page table entries are flushed from the TLB, which amounts to a non-trivial overhead on Pentium IV processors.

The second section of the table compares the pipe RPC mechanism in UNIX systems, with comparable functionality to perform a round-trip transfer of one byte of data between a pair of Hijack guests via the executive. As can be seen, Hijack RPC between two guests executes approximately 36% faster than traditional pipes (i.e., 9250 cycles versus 14500), even with the added cost of manipulating global bits when transitioning between guests. Additionally, traditional pipe latency is not predictable as the kernel scheduler can introduce delays due to multiple tasks in its run queue. In contrast, the real-time executive is capable of defining its own RPC semantics between guests to ensure predictable communication. In general, Hijack does not induce unacceptable performance penalties due to protected interposition and the communication it entails.

TLB Costs. To complement the above benchmarks, Figure 4 depicts a series of simple remote procedure call (RPC) tests that measure the impact of the TLB on performance. The working set size of a Hijack guest or, equivalently, Linux client process is varied in terms of the number of instruction and data pages it uses. To do this, we make jumps to instructions, or reads from addresses, that are 4160 bytes apart to minimize cache interference. This is similar to our own work on ULS [23], and that of "small spaces" [20] in L4. For Hijack guests, RPC involves exchanging four bytes with a function in the executive, while an equivalent Linux pipe experiment exchanges the same data between two separate processes.

As can be seen from Figure 4, the control path using Hijack is far less costly in terms of clock cycles, because only the pages of the executive need to be mapped and then flushed during the round-trip communication with a guest. In contrast, a Linux pipe will involve flushing the entire TLB when switching between processes (triggered by a modification to `CR3` to reference new page table mappings). This is emphasized by the observed TLB misses (for user-level pages) as a function of instruction working set size in Figure 4(c). Shown in this figure is the impact on the TLB when making a conventional system call (here using `getppid()`) down into the host kernel. This reference line shows the LRU replacement policy on the instruction TLB, which is clearly noticeable from the step increase in misses around the i-TLB capacity (of 128 page entries on the Pentium 4). For the Hijack RPC, there are 6 TLB misses to flush the executive on the return to a guest. Only when the i-TLB is filled to capacity do we observe additional misses using Hijack and even then the increase is at a rate of about 2 misses per additional page until the working set is almost double the i-TLB capacity. We speculate that global pages are being evicted from the i-TLB according to some different policy than LRU but, nonetheless, Hijack never incurs more TLB costs than conventional pipes. These experiments clearly demonstrate the benefit of our scheme using global bits to avoid flushing corresponding guest pages from the TLB.

3.2 Kernel Event Notification

In addition to interposition, the notification of kernel events to Hijack must be efficient and predictable. For instance, the predictable notification that a given amount of time has expired (commonly manifested as a clock-tick) is necessary for any scheduling algorithm to be performed in the executive. Additionally, in order to respond to hardware events such as packet reception or sensor information in a bounded amount of time, the executive must be notified of these events predictably. Mechanisms such as signals, and traditional event notification system calls (`select`, `poll`, etc...) are not predictable in general-purpose systems, as they are subject to non-real-time scheduling delays of corresponding processes. Because the Hijack thread is the highest priority in the system it is assured of predictable event notifications. To demonstrate this, a task is set to receive a SIGALRM signal as frequently as possible using the `setitimer` system call. A number of CPU-bound tasks are run in the background, and the average inter-arrival time of each delivered signal is measured with the time-stamp counter provided by Pentium IV processors. Ideally, the inter-arrival time will be every clock tick on the host system, which is every 10 milliseconds (as configured in the Linux kernel).

Figure 5 shows the average signal inter-arrival time for

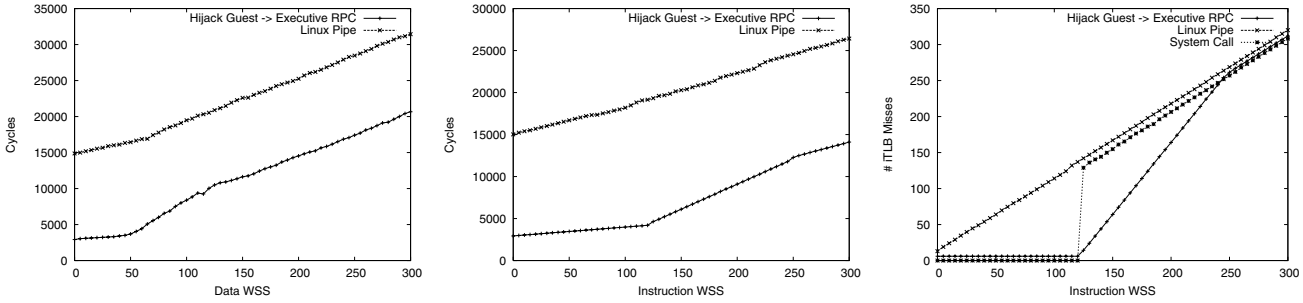

Figure 4. TLB Impact on RPC.

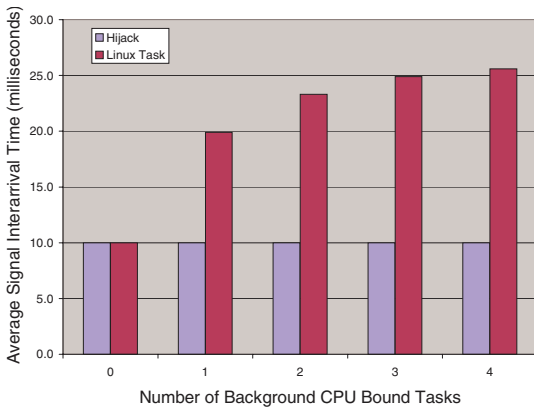

Figure 5. Predictability of kernel events.

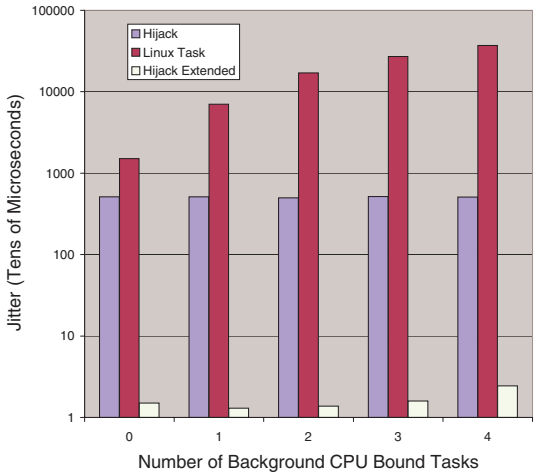

Figure 6. `nanosleep` **predictability.**

two situations. In the first case (labeled "Hijack"), the signal is received by the Hijack executive itself, which uses its own round-robin scheduler to order the execution of 0 to 4 background guests. In the second case (labeled "Linux Task"), a signal is delivered to a normal Linux task in the presence of 0 to 4 background tasks all scheduled in a round-robin manner by the host kernel. As can be seen, Hijack maintains a predictable event reception inter-arrival time of 10 milliseconds, while the default Linux case yields increasing inter-arrival times as more background tasks are scheduled by the host.

Predictable kernel event notification allows the timely processing of external events. Figure 6 shows the performance of a `nanosleep` system call, that allows a task to sleep for a specified number of nanoseconds. The performance of `nanosleep` can have a significant impact on time-sensitive applications, and researchers have gone to great lengths, such as modifying COTS systems, to achieve a more predictable implementation [6].

The prevalent cause of unpredictability in `nanosleep` is due to two factors. First, like other event-notification mechanisms, the notification of a sleep timer expiration is subject to the system scheduler which, in COTS systems, is often not ideal for real-time tasks. Second, for sleep periods that expire before the next system clock tick, the kernel will wake up the task too late, at the time of that clock tick [2]. This can lead to `nanosleep` having a practical granularity of the frequency of the system clock. Using Hijack, and the predictable delivery of clock ticks from the kernel, we implement two improvements to the traditional `nanosleep` call. First, the `Hijack` version simply alleviates the scheduling problem by always delivering an expired sleep period notification to the executive regardless of which guest is currently loaded. From this point, the executive can wake up the specific guest whose nanosleep duration has expired. This might be an appropriate policy for systems where predictable sleeping periods are more important than task fairness. Second, the `Hijack Extended` case attempts to (in addition to the changes mentioned above) provide a practical sleeping granularity significantly less than a clock tick. This case will rely on predictable clock ticks for detecting sleeps expiring after the next tick, while the executive busy waits when expiration times are less than a clock tick. This is not a general solution as the time spent busy waiting might be significant and, if there are other tasks in the system, they will not be al-

[2] With the exception of modern Linux systems which will busy wait for expiration times less than 2 microseconds.

lowed to execute while the executive busy waits. Thus overall throughput can decrease, while predictability improves. Hijack allows the redefinition of such system policies if it is appropriate for the task at hand.

Figure 6 presents the results using `nanosleep` in different systems. In each case, a task executes the `nanosleep` system call a number of times starting with a sleep value of 100 nanoseconds and increasing by half a millisecond each call up until a sleep for one tenth of a second. A time stamp is recorded before the system call and immediately after, and the actual amount of time slept is compared to the requested sleep period and the difference is recorded as the jitter. This setup is used in the two cases described above, and also with a single Linux task without the Hijack system. The number of background CPU-bound tasks (guests in Hijack) is altered to trigger scheduler interference, and the average jitter is reported. One can see that the jitter for `Hijack` case does not vary significantly as the number of background tasks changes. This is because the executive defines the policy for predictable sleeping, and is therefore able to wakeup and immediately execute a sleeping guest when its sleep period ends (albeit at the granularity of a clock tick). Hijack allows the system policies to be redefined as in the `Hijack Extended` case where the executive busy waits for waiting periods less than a clock tick. The jitter in this case is orders less than the others because its sleeping granularity is significantly smaller than the other cases.

Note that the functionality presented in this section can be implemented in a number of alternative ways. However, the purpose of these experiments is to demonstrate the predictability of kernel event notifications and the functional capabilities of the Hijack software architecture. Hijack is able to support predictable functionality in a protected way that could not easily be achieved either by library-based services linked into the address spaces of untrusted application-level processes, or by convoluted manipulation of underlying system services using conventional system calls. In effect, services implemented within Hijack's executive hide these issues from the guest applications.

3.3 Using Hijack for QoS based Resource Allocation

This section demonstrates the ability of Hijack to efficiently and predictably manage resources based on QoS constraints. Multiple tasks concurrently process different streams incoming from the network, so there is competition for system resources such as CPU cycles. There are four tasks in the system, whereby $Task0$ is reserved a throughput of 35,000 packets/second, $Task1$ is reserved 20,000 packets/second, $Task2$ is reserved 10,000 packets/second and $Task3$ is best effort, and should consume any surplus resources when all reservations are met.

The experiments involve four hosts, B, C, D and E sending UDP packet streams of 42,000 packets/second to the four tasks on a destination host A. Each packet is 16 bytes and consists of a serial number which is used to determine if and when packets are dropped, and also a timestamp when the corresponding packet is sent into the network. Immediately upon reception of a packet at host A by the corresponding task, another cycle timestamp is recorded. Contiguous timestamps within packets from the same stream yield the inter-send time for those packets, whereas contiguous timestamps recorded upon packet reception yield inter-arrival times. The difference between the inter-send time and the inter-arrival time for two contiguously received packets is what we define as the jitter for those packets. Each task receives packets, stores them, and once 15,000 have been received, computes the total number of packets delivered per second, amount dropped, and the average and maximum jitter. Reservations are currently made in packets/second, but it would be trivial to extend the reservations to bits/second.

```
// init_tokens(task) is an immutable initial
//             token allocation for a task

// curr_tokens(task) is the current number
//                   of tokens for a task

// 'tasks' array is sorted from highest QoS to lowest

main_event_loop () {
   next = NULL;

   select on the file descriptors for each task;

   if (timing period has expired) {
      for (each task in tasks)
           curr_tokens(task) = init_tokens(task);
   }

   for (each task in tasks) {
      if (select indicated that task has data) AND
         (curr_tokens(task) > 0) {
           next = task;
           break;
      }
   }
   if (next == NULL)
      next = best_effort_task;
   execute next;
}

guest_syscall_read(guest_fd, guest_buf, guest_size) {
   fd = translate_to_host_fd(guest_fd);

   loop until (read doesn't return data) OR
              (curr_tokens(task) == 0) {
      read(fd, guest_buf, guest_size); //nonblocking
      curr_tokens(task)--;
   }
}
```

Figure 7. Hijack I/O scheduling algorithm.

Here, we segregate the experiment into three distinct implementations. First, the `same prio` case: each and every

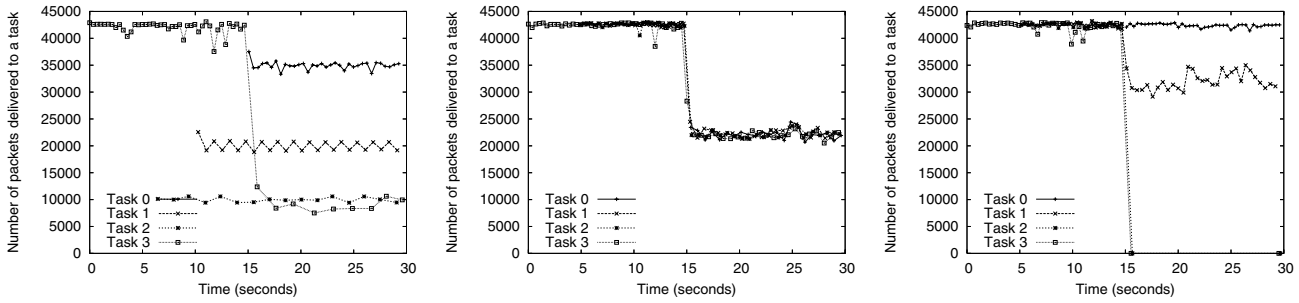

Figure 8. Packets delivered using (a) Hijack, (b) same priority tasks, and (c) different priority tasks.

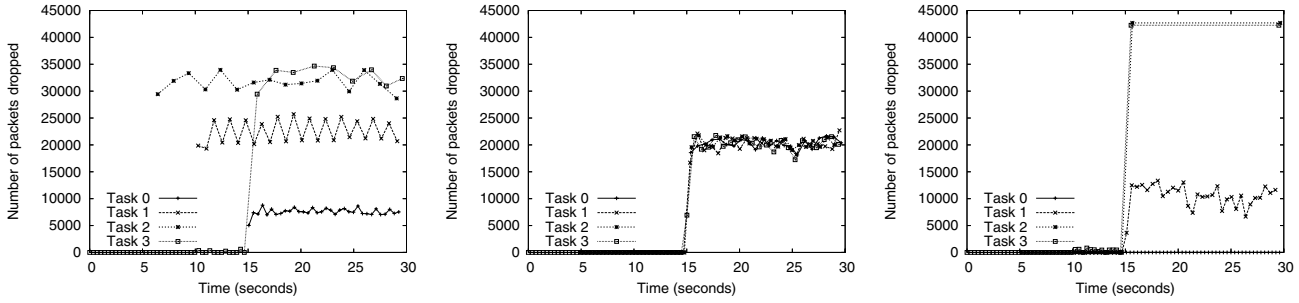

Figure 9. Packets dropped using (a) Hijack, (b) same priority tasks, and (c) different priority tasks.

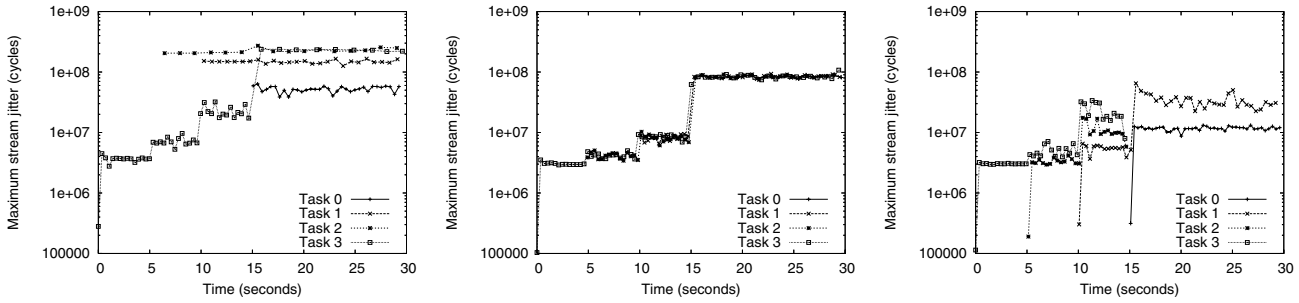

Figure 10. Max. stream jitter using (a) Hijack, (b) same priority tasks, and (c) different priority tasks.

Linux task is set to run using the SCHED_RR scheduling policy with the same fixed priority, with each task servicing a data-stream. Second, the `diff prio` case is similar to `same prio`, except that $Task0$ has the highest priority, $Task1$ has the second highest, $Task2$ has the third highest, and $Task3$, the best effort task, has the lowest priority of the four. This assignment of priorities attempts to approximate the QoS requirements of the tasks, by reflecting their relative importance. Third, the `hijack` case, each task is run as a guest in a Hijack execution environment with a real-time executive scheduling the guests appropriately based on their QoS requirements.

The executive runs a QoS-aware algorithm simplified in Figure 7. Tasks are allocated a number of tokens commensurate with the number of messages they need to receive per second. The tokens allocated to a task are refreshed periodically, at 10 times per second. Tasks are scheduled based on two factors: (1) the presence of available packets to process for that task's stream, and (2) the task's QoS. In the algorithm, a `task`'s tokens are specified by the `init_tokens(task)` value. As tasks execute, their tokens are depleted. When a task is executed, it will read from its corresponding packet stream, and the read service request will be interposed by the executive. The executive will continuously read packets from the corresponding data-stream into the guest's buffer using non-blocking I/O until there is no pending data, the guest has run out of buffer space, or the task runs out of tokens.

In each of the experiments, the best effort task, $Task3$, begins (at t=0) receiving its data-stream. At five second intervals, $Task2$, $Task1$, and $Task0$, respectively, start receiving data. This leads the system from an under-load to overload situation. Figure 8 depicts the number of packets per second delivered to each task (for, left-to-right:

`hijack`, `same prio`, and `diff prio` cases). Likewise, Figure 9 plots the amount of packets dropped per second for each task.

In both cases that use task priorities within Linux, there is no notion of QoS-aware I/O, so no reservations are made for the appropriate tasks. In hijack, the appropriate amount of I/O processing is performed for each task to maintain its reservation, while the best effort task uses any surplus resources. When using the same priorities for tasks there is no differentiation between delivery and drop rates of packets across the four corresponding streams. Contrarily, when using different task priorities, the two tasks with highest priorities receive a disproportionate degree of service relative to the two least important tasks. Effectively, $Task0$ and $Task1$ are being allocated resource shares above their required levels, while starving $Task2$ and $Task3$.

The final set of graphs in Figure 10 show the maximum jitter in the delivery of data streams. The `same prio` case reflects the same quality of service to all four streams and, consequently, results in the same jitter in packet reception across all four tasks. While `hijack` leads to greater jitter for lower priority tasks, $Task0$ actually experiences less jitter and, hence, more favorable service than any of the tasks in the `same prio` case. In contrast, the `diff prio` case maintains relative low maximum jitter for the two highest priority tasks which receive service, at the cost of infinite delay (and, therefore, extreme jitter) for the two least important tasks.

Our results for average jitter show similar patterns to those for maximum jitter and have therefore been omitted, for brevity. We acknowledge that hijack results in more jitter for the two lowest QoS tasks, compared to the `same prio` case, but this is consistent with the relative service rates of those tasks. Moreover, only `hijack` is able to achieve the desired service rates for the QoS-constrained tasks.

In summary, Hijack provides the basis for implementing efficient and predictable QoS-constrained resource and task management policies. The above experiments demonstrate that QoS based decisions regarding resource usage can be manifested in Hijack to provide more predictable service guarantees.

4. Related Work

In the operating systems community, there have been various research endeavors ranging from system structure to extensibility that relate to our work. Micro-kernels such as Mach [1] and L4 [9, 10] originally considered the idea of isolating all but the most basic service abstractions outside the core kernel protection domain. We adhere to this philosophy with our work, by providing an execution environment as part of Hijack that resides outside the base kernel. The benefit of this approach, for us, is not only in terms of the added safety due to isolating the kernel from untrusted application-specific services, but also in that we can leverage a largely unmodified COTS system including all of its pre-written services and low-level device drivers.

Extensible systems such as VINO [18] and SPIN [3] are designed from the ground-up to support application-specific services within the kernel, but such untrusted code is prevented from jeopardizing the integrity of the system using techniques such as software-based fault isolation [22] and/or type-safe languages [7, 24]. Our approach differs, in that we wish to support application-specific real-time services and predictable service invocation *outside* the base kernel. To that end, our work is an attempt to empower off-the-shelf systems with features to support predictable and application-configurable real-time services.

Using off-the-shelf systems in real-time systems has been the focus of a number of research groups for some years [13, 14, 17, 16, 19, 11]. 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. In this way, non-real-time tasks can be deferred while hard real-time tasks execute with bounded and low latency. Such approaches have proved to be very effective for real-time computing but, for the most part, these approaches are "shared address space" systems, in which real-time tasks execute within the kernel address space. As stated earlier, our approach isolates real-time tasks and services outside the core kernel. Since Hijack services may make use of the underlying services of the base kernel, in a manner similar to KURT [19] and Linux/RK [11], there is possibility for some unpredictability due to e.g., interrupts being disabled in host device drivers for synchronization purposes. In such cases, Hijack services would be most suitable for soft, as opposed to hard, real-time computing supported by approaches such as RTLinux. While we have chosen to implement Hijack on a COTS Linux system, it would be possible to provide a similar infrastructure on systems such as RTLinux, thereby allowing Hijack services to rely on very predictable services of the underlying kernel. In effect, reliance on any underlying services of the host kernel dictates the suitability of Hijack to hard or soft real-time applications.

Given that we now have the ability to insert code between applications and the kernel we have, in effect, an efficient and predictable execution environment for middleware services. Specifically, systems such as Tao [12] that are built using CORBA must isolate real-time services in process-private address spaces, that are subject to non-real-time scheduling and IPC mechanisms of the host system.

Other related work includes the Drops system [10]. Drops has been used as the basis for L4RTL, a real-time variant of the L4 micro-kernel, that isolates real-time tasks,

timesharing tasks, and other non-real services in separate address spaces. While this approach is more endearing to our philosophy of isolating application-specific real-time services outside the core kernel, it is more of an attempt to show how micro-kernels (built from the base hardware up) can be used for predictable computing. Additionally, the "small spaces" [20] technique in L4 uses a combination of page global bits and segmentation to reduce the cost of IPC between micro-kernel services. We, on the other hand, are trying to use the basic services of underlying off-the-shelf systems with interposition [5, 8] features to provide a real-time execution environment outside the core kernel, without relying on special hardware support such as segmentation, or specialized OSes. This, in turn, differentiates our work from other operating systems purposefully designed for real-time computing, such as QNX, VxWorks, as well as those systems designed around special hardware protection features [4].

In many ways, our Hijack executive has similarities to that of a virtual machine monitor [2], which has the ability to intercept the execution of guest operating systems and their applications, thereby ensuring physical resources are correctly multiplexed between all supported virtual machines. Hijack intercepts system service requests, rather than machine instructions, and provides a basis for predictable application-specific services. This is similar to User-Mode Linux [21] (UML) that originally used `ptrace` to virtualize the system call interface. As shown in Section 3.1, `ptrace` is a relatively inefficient method to support interposition, as it was originally intended as a debugging facility. A more recent version of UML requires host kernel modifications, so that guest Linux virtual machines can execute with improved efficiency. However, UML makes no attempt to establish an execution environment with which to support real-time application-specific services, which is the basis of our work. Moreover, Hijack avoids changes to the underlying kernel to achieve its interposition capabilities.

5. Conclusions and Future Work

Commodity systems are appealing for real-time tasks as they provide a common code base and a well-tested and supported environment, lowering maintenance costs and easing development. However, a semantic gap exists between the needs of real-time applications and the capabilities of COTS systems. We address this with a novel mechanism for "hijacking" an unpredictable kernel, using a real-time executive deployed at user-level, along with techniques to interpose code on system calls and hardware interrupts. We define an interface which allows the executive to manipulate both address spaces and register states of all guests under its control. This allows for services (e.g., for real-time scheduling) to be deployed in the Hijack executive, which assumes precedence over all other control flows in the system.

We demonstrate multiple applications of this mechanism that provide more predictable system behavior and manage system resources, while explicitly taking into account application-level QoS metrics. This is achieved without making cumbersome changes to the core kernel of the underlying COTS system. Experimental results show that for a real-time streaming application, Hijack is able to more predictably manage resource usage amongst competing tasks compared to alternative techniques using pre-existing policies within COTS systems.

Future work involves studying novel services and policies interposed on hardware interrupts, in addition to those interposed on system calls and page faults. Additionally, we intend to investigate strategies for deploying Hijack on multi-core and multi-processor platforms. While we already have ideas how to do this, we will investigate techniques that use shadow pages and inter-processor-interrupts to manipulate per-processor TLBs. Finally, methods to avoid "QoS crosstalk" [2] will be considered by addressing factors such as resource sharing (of caches and synchronization objects, for example) amongst non-real-time background processes and Hijack guests. Our plan is to have Hijack and our ongoing ULS [23] code available for download, so that it can be used by the systems community.

6. Acknowledgments

We would like to thank Doug Niehaus and the anonymous reviewers, who together have helped improve the quality of this work.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Summer USENIX Conference*, pages 93–112, 1986.

[2] 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 *Symposium of Operating System Principles*, 2003.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, and B. E. Chambers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, 1995.

[4] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceeedings of the ACM Symposium on Operating Systems Principles*, pages 140–153, 1999.

[5] D. P. Ghormley, S. H. Rodrigues, D. Petrou, and T. E. Anderson. Interposition as an operating system extension mechanism. Technical Report CSD-96-920, University of California, Berkeley, September 1997.

[6] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole. Supporting time-sensitive applications on a commodity os. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[7] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, June 2002.

[8] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 80–93, December 1993.

[9] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. ACM, December 1995.

[10] F. Mehnert, M. Hohmuth, and H. Hartig. 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.

[11] 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.

[12] I. Pyarali, D. C. Schmidt, and R. Cytron. Achieving end-to-end predictability of the tao real-time corba orb. In *Proceedings of the 8th IEEE Real-Time Technology and Applications Symposium*, San Jose, CA, September 2002.

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

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

[15] J. Regehr and J. Stankovic. Hls: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, December 2001.

[16] Real-Time Application Iinterface: http://www.rtai.org.

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

[18] C. Small and M. I. Seltzer. A comparison of OS extension technologies. In *USENIX Annual Technical Conference*, pages 41–54, 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] V. Uhlig, U. Dannowski, E. Skoglund, A. Haeberlen, and G. Heiser. Performance of address-space multiplexing on the Pentium. Technical Report 2002-1, University of Karlsruhe, Germany, 2002.

[21] The user-mode linux kernel home page: http://user-mode-linux.sourceforge.net/.

[22] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[23] 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.

[24] R. West and G. Wong. Cuckoo: a language for implementing memory- and thread-safe system services. In *Proceedings of the International Conference on Programming Languages and Compilers*, June 2005.