# Predictable Interrupt Management and Scheduling in the Composite Component-based System *

Gabriel Parmer and Richard West

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

## Abstract

*This paper presents the design of user-level scheduling hierarchies in the Composite component-based system. The motivation for this is centered around the design of a system that is both dependable and predictable, and which is configurable to the needs of specific applications. Untrusted application developers can safely develop services and policies, that are isolated in protection domains outside the kernel. To ensure predictability, Composite needs to enforce timing control over user-space services. Moreover, it must provide a means by which asynchronous events, such as interrupts, are handled in a timely manner without jeopardizing the system. Towards this end, we describe the features of Composite that allow user-defined scheduling policies to be composed for the purposes of combined interrupt and task management. A significant challenge arises from the need to synchronize access to shared data structures (e.g., scheduling queues), without allowing untrusted code to disable interrupts or use atomic instructions that lock the memory bus. Additionally, efficient upcall mechanisms are needed to deliver asynchronous event notifications in accordance with policy-specific priorities, without undue recourse to schedulers. We show how these issues are addressed in Composite, by comparing several hierarchies of scheduling polices, to manage both tasks and the interrupts on which they depend. Studies show how it is possible to implement guaranteed differentiated services as part of the handling of I/O requests from a network device while avoiding livelock. Microbenchmarks indicate that the costs of implementing and invoking user-level schedulers in Composite are on par with, or less than, those in other systems, with thread switches more than twice as fast as in Linux.*

*This material is based upon work supported by the National Science Foundation under Grant Numbers 0615153 and 0720464. 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

As software complexity increases, it becomes increasingly more difficult to verify that system and application-level code will behave correctly under various operating conditions. Likewise, the need for fault isolation to be a central focus of system design is becoming increasingly important. Safety critical systems require both dependability and predictability, with emphasis placed on timely execution and fault tolerance. To limit the scope of impact of potentially faulty or misbehaving software on the overall system, it makes sense to encapsulate logical units of functionality in separate components mapped to their own protection domains. Likewise, untrusted software, or software developed by third-parties, needs to be isolated from the trusted kernel protection domain. Component-based systems [7, 22] and micro-kernels [16, 1] provide solutions to the separation of services and applications. Extensibility is integral to these system designs, in that application-specific services can be added to a system, and mapped to their own logical protection domains, thereby bridging the "semantic gap" between application needs and existing service provisions. The challenge, however, is to ensure the interaction between such services remains predictable and efficient.

This paper focuses on the design of predictable and efficient user-level services in our Composite component-based system. In particular, we show how a hierarchy of component-based schedulers can be supported with our system design. By isolating such services in user-space, we avoid potentially adverse interactions with the trusted kernel protection domain, that could otherwise render the system inoperable, or could lead to unpredictability. We show how a series of schedulers can be composed to manage both the handling of interrupts as well as conventional threads of execution. Specifically, in situations where threads make I/O requests on devices that ultimately respond with interrupts, we ensure that interrupt handlers are scheduled in ac-

cordance with the urgency and importance of the threads that led to their occurrence. In essence, there is a *dependency* between interrupt and thread scheduling that is not adequately solved by many existing operating systems [34], but which is addressed in our component-based system.

While micro-kernels offer a means by which new services can be deployed at user-level, inter-process communication (IPC) is still mediated by the trusted kernel. Such IPC mechanisms require thread switching, often involving scheduling decisions. In early micro-kernel designs, IPC costs were deemed prohibitive, but more recent systems have addressed such costs, often by using hardware-specific features [16]. In Composite, communication between components is carried out by thread migration [10] to avoid scheduling costs during IPC. A thread executing in one component may communicate with, and continue execution in, another component. This design also avoids complications concerning thread synchronization on IPC [29, 26].

User-level schedulers introduce design challenges due to the need to make scheduling decisions both predictably and efficiently. Synchronization around scheduler data-structures is complicated by the fact that interrupts are not allowed to be disabled. Preventing interrupts from being disabled is required for Composite to be predictable. Similarly, for efficiency, it is undesirable to issue kernel requests to access synchronization objects such as semaphores, especially if the cost of kernel-user transitions is high. Worse still, kernel-provided synchronization mechanisms such as semaphores may yield deadlock or livelock situations on access to user-level scheduling structures. As an example, suppose $\tau_1$ is preempted while holding a semaphore, $S$, for a scheduling queue. An upcall thread, $\tau_2$ (e.g., for a timer interrupt), tries to access $S$ and cannot proceed. At this point, the kernel needs to know which thread to schedule and upcalls into the user-level scheduler in $\tau_2$'s context, with yet another attempt to acquire $S$. Thus, neither $\tau_1$ nor $\tau_2$ are able to make effective progress. Other problems include the use of atomic instructions which can lock the memory bus, causing undue latencies. We address these practical issues in Composite using an optimistic synchronization mechanism, based on "restartable atomic sequences" [6] and inspired by futexes [12]. During typical execution, synchronization is ensured without relying on atomic instructions or kernel invocations. This utility is used both for inter-thread synchronization in schedulers and for synchronization between threads and a non-preemptive kernel, and does not require a kernel-resident scheduler.

Another significant challenge is the predictable scheduling and accounting of asynchronous events, such as interrupts. One of the goals of Composite is to associate interrupts with their own threads of execution, for the purposes of scheduling. However, we wish to avoid invocations of user-level schedulers every time an asynchronous event occurs, as this would be too expensive. Consequently, Composite provides a mechanism by which asynchronous threads can be executed in accordance with their scheduling constraints without direct invocation of user-level schedulers.

In Composite, when an interrupt occurs it is initially trapped by the base kernel. From there, an *upcall* is made into a user-space component to handle the interrupt. Associated with each such upcall is a *brand*, that identifies user-level scheduler information for the interrupts. Additionally, brands record the sequence of components that are to be executed by the upcall in response to an asynchronous event.

In the following sections we elaborate on the design details within Composite, focusing on the design of hierarchical component services for predictable and efficient scheduling and interrupt management. Section 2 describes in further detail some of the design challenges of Composite, including the implementation of hierarchical schedulers. This is followed by an experimental evaluation in Section 3. Related work is discussed in Section 4 while a summary of conclusions and future work are presented in Section 5.

## 2 Composite Component-based Scheduling

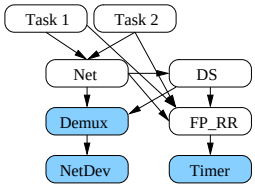


**Figure 1. An example component graph.**

In Composite, a system is constructed from a collection of user-level components that define the system's policies. These components communicate via thread migration and compose to form a graph as depicted in Figure 1. Edges imply possible invocations. Here we show a simplified component graph with two tasks, a networking component, a fixed priority round-robin scheduler, and a deferrable server. As threads make component invocations, the system tracks their progress by maintaining an *invocation stack*, as shown in Figure 2. In this example, the downward control flow of a thread proceeds through `A`, `B`, `C` and `D`. Each of these invocations is reflected in its execution stack. On return from `D` and `C`, the thread pops components off of its execution stack, and invokes `E`. Its invocation stack after this action is shown with the dotted lines.

One of the goals of Composite is to provide a base system that is configurable for the needs of individual applications. However, for performance isolation, it is necessary that global policies maintain system-wide service guarantees across all applications. Consequently, we employ a hierarchical scheduling scheme [24] whereby a series of suc-

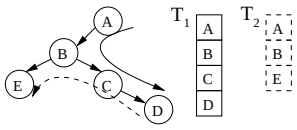


**Figure 2. Thread execution through components.**

cessive schedulers are composed in a manner that maintains control by more trusted schedulers, while still allowing policy specialization for individual applications. In addition to traditional notions of hierarchical scheduling, we require that parent schedulers do not trust their children, and are isolated from their effects. In this way, the affects of faulty or malicious schedulers are restricted to their subtree in the scheduling hierarchy. By comparison, scheduler activations [3] are based on the premise that user-level schedulers interact with a more trusted kernel scheduler in a manner that cannot subvert the kernel scheduler. Composite adopts a mechanism that generalizes this notion to a full hierarchy of schedulers that all exist at user-level.

Composite exports a system call API for controlling the scheduler hierarchy. This enables the construction of a recursive structure of schedulers whereby more trusted (parent) schedulers *grant* scheduling privileges to their children. Likewise, schedulers have the ability to *revoke*, transitively, all such scheduling permissions from their children. To bootstrap the system, one scheduler is chosen to be the root scheduler. Creating new child schedulers is done with feedback from abstract application requirements [14], or via application specified policies.

| | |
|---|---|
| `COS_SCHED_PROMOTE_SCHED` | promote component to be a child scheduler |
| `COS_SCHED_DEMOTE_SCHED` | remove child subtree's schedulers privileges |
| `COS_SCHED_GRANT_THD` | grant scheduling privileges to child scheduler for a specific thread |
| `COS_SCHED_REVOKE_THD` | revoke scheduling privileges to a child scheduler's subtree for a specific thread |
| `COS_SCHED_SHARED_REGION` | specify a region to share with the kernel |
| `COS_SCHED_THD_EVT` | specify an event index in the shared region to associate with a thread |

**Table 1.** `cos_sched_cntl` **options.**

A component that has been promoted to scheduler status has access to the `cos_sched_cntl(operation, thd_id, other)` system call. Here `operation` is simply a flag, the meaning of which is detailed in Table 1, and `other` is either a component id, or a location in memory, depending on the operation. In a hierarchy of scheduling components only the root scheduler is allowed to create threads. This restriction prevents arbitrary schedulers from circumventing the root scheduler for allocating kernel thread structures. Thus, it is possible for the root scheduler to implement thread creation policies (e.g. quotas) for specific subsystems or applications. Threads are both created and passed up to other components via the `thd_id cos_thd_cntl (component_id, flags, arg1, arg2)` system call. To create a new thread, the root scheduler makes a system call with the `COS_THD_CREATE` flag. If a non-root scheduler attempts to create a new thread using this system call an error is returned. Threads all begin execution at a specific upcall function address added by a Composite library into the appropriate component. Such an invocation is passed three arguments: the reason for the upcall (in this case, because of thread creation) and the user-defined arguments `arg1` and `arg2`. These are used, for example, to emulate `pthread_create` by representing a function pointer and the argument to that function.

In Composite, each kernel thread structure includes an array of pointers to corresponding schedulers. These are the schedulers that have been granted scheduling privileges over a thread via `cos_sched_cntl(COS_SCHED_GRANT_THD,...)`. The array of pointers within a thread structure is copied when a new thread is created, and is modified by the `cos_sched_cntl` system call. Certain kernel operations must traverse these structures and must do so with bounded latency. To maintain a constant overhead for these traversals, the depth of the scheduling hierarchy in Composite is limited at system compile time.

## 2.1 Implementing Component Schedulers

Unlike previous systems that provide user-level scheduling [11, 4, 30], the operation of blocking in Composite is not built into the underlying kernel. This means that schedulers are able to provide customizable blocking semantics. Thus, it is possible for a scheduler to allow arbitrary blocking operations to time-out after waiting for a resource if, for example, a deadline is in jeopardy. In turn, user-level components may incorporate protocols to combat priority inversion (e.g., priority inheritance or ceiling protocols [27]).

Not only do schedulers define the blocking behavior, but also the policies to determine the relative importance of given threads over time. Given that schedulers are responsible for thread blocking and prioritization, the interesting question is what primitives does the kernel need to provide to allow the schedulers to have the greatest freedom in policy definition? In Composite, a single system call is provided, `cos_switch_thread(thd_id, flags)` that permits schedulers with sufficient permissions to dispatch a specific thread. This operation saves the current thread's registers into the corresponding kernel thread structure, and restores those of the thread referenced by `thd_id`. If the next thread was previously preempted, the current protection domain (i.e. page-table information) is switched to that of the component in which the thread is resident.

In Composite, each scheduling component in a hierarchy

can be assigned a different degree of trust and, hence, different capabilities. This is related to scheduler activations, whereby the kernel scheduler is trusted by other services to provide blocking and waking functionality, and the user-level schedulers are notified of such events, but are not allowed the opportunity to control those blocked threads until they return into the less trusted domain. This designation of duties is imperative in sheltering more trusted schedulers from the potential ill-behavior of less trusted schedulers, increasing the reliability of the system. An example of why this is necessary follows: suppose a network device driver component requests the root scheduler to block a thread for a small amount of time, until the thread can begin transmission on a TDMA arbitrated channel. If a less trusted scheduler could then restart that thread before this period elapsed, it could cause detrimental contention on the channel. The delegation of blocking control to more trusted schedulers in the system must be supported when a hierarchy of schedulers is in operation. To allow more trusted schedulers to make resource contention decisions (such as blocking and waking) without being affected by less trusted schedulers, a flag is provided for the `cos_switch_thread` system call, `COS_STATE_SCHED_EXCL`, which implies that only the current scheduler and its parents are permitted to wake the thread that is being suspended.

## 2.2 Brands and Upcalls

Composite provides a notification mechanism to invoke components in response to asynchronous events. For example, components may be invoked in response to interrupts or events similar to signals in UNIX systems. In many systems, asynchronous events are handled in the context of the thread that is running at the time of the event occurrence. In such cases, care must be taken to ensure the asynchronous execution path is reentrant, or that it does not attempt to block on access to a lock that is currently being held by the interrupted thread. For this reason, asynchronous event notifications in Composite are handled in their own thread contexts, rather than on the stack of the thread that is active at the time of the event. Such threads have their own priorities so they may be scheduled in a uniform manner with other threads in the system. Additionally, a mechanism is needed to guide event notification through multiple components. For example, if a thread reads from a UDP socket, and an interrupt spawns an event notification, it may be necessary to traverse separate components that encompass both IP and UDP protocols.

Given these constraints, we introduce two concepts: *brands* and *upcalls*. A brand is a kernel structure that represents (1) a context, for the purposes of scheduling and accounting, and (2) an ordered sequence of components that are to be traversed during asynchronous event handling. Such brands have corresponding priorities that reflect the urgency and/or importance of handling a given event notification. An upcall is the active entity, or thread associated with a brand that actually executes the event notification. Brands and upcalls are created using the `cos_brand_cntl` system call. In the current implementation, only the root scheduler is allowed to make this system call as it involves creating threads. The system call takes a number of options to create a brand in a specific component, and to add upcalls to a brand. Scheduling permissions for brands and upcalls can be passed to child schedulers in the hierarchy in exactly the same fashion as with normal threads. An upcall associated with a given brand is invoked by the `cos_brand_upcall(brand_id, flags)` system call, in a component in which that brand has been created.

Figure 3 depicts branding and upcall execution. A thread traversing a specific path of components, `A, B, C, D`, requests that a brand be created for invocation from `C`: $T_B$ = `cos_brand_cntl(COS_BRAND_CREATE_BRAND, C)`. Thus, a brand is created that records the path already taken through components `A` and `B`. An upcall is added to this brand with `cos_brand_cntl(COS_BRAND_CREATE_UPCALL,` $T_B$`)`. When the upcall is executed, component `B` is invoked, as depicted with the coarsely dotted line. This example illustrates a subsequent component invocation from `B` to `E`, as depicted by the finely dotted line.

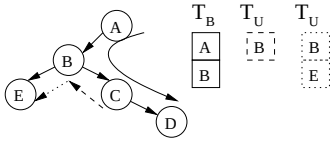


**Figure 3. Branding and upcall execution.**

When an upcall begins execution in a component, it invokes the generic upcall function added to the component via the Composite library. If an event occurs that requires the execution of an upcall for the same brand as an active upcall, there are two options. First, if there is an inactive upcall associated with a brand, then the inactive upcall can be executed immediately to process the event. The precise decision whether the upcall is immediately executed depends on the scheduling policy. Second, if all upcalls associated with a brand are active, then a brand's pending count of events is incremented. When an upcall completes execution and finds that its brand has a positive event count, the count is decremented and the upcall is re-instantiated.

Brands and upcalls in Composite satisfy the requirements for asynchronous event notification, but an important aspect is how to efficiently and predictably schedule their corresponding threads. When the execution of an upcall is attempted, a scheduling decision is required between the currently running thread and the upcall. The scheduler

that makes this decision is the *closest common* scheduler in the hierarchy of both the upcall and the currently executing thread. Additionally, when an upcall has completed execution, assuming its brand has no pending notifications, we must again make a scheduling decision. This time the threads that are candidates for subsequent execution include: (1) the thread that was previously executing when the upcall occurred, (2) any threads that have been woken up by the upcall's execution, and (3) any additional upcalls that occurred in the meantime (possibly due to interrupts), that were not immediately executed. At the time of this scheduling decision, one option is to upcall into the root scheduler, notifying it that the event completed. It is then possible for other schedulers in the hierarchy to be invoked. Unfortunately, invoking schedulers adds overhead to the upcall, and increases the response time for event notification. We, therefore, propose a novel technique in which the schedulers interact with the kernel to provide hints, using *event structures*, about how to perform subsequent scheduling decisions without requiring their invocation during upcall execution. This technique requires each scheduler in the system to share a private region with the kernel. This region is established by passing the `COS_SCHED_SHARED_REGION` flag to the `cos_sched_cntl` system call detailed in Table 1.

Corresponding threads under the control of a given scheduler are then associated with an event structure using the `COS_SCHED_THD_EVT` flag. Each of these event structures has an *urgency* field, used for priority-based scheduling. Depending on the policy of a given scheduler, urgencies can be dynamic (to reflect changing time criticality of a thread, as in the case of a deadline) or static (to reflect different degrees of importance). Numerically lower values for the urgency field represent higher priorities relative to other threads. Within the event structure, there is also a flag section to notify schedulers about the execution status of the thread associated with the event. This is relevant for inactive upcalls, as they are not currently schedulable. The last field of the event structure is an index pointer used to maintain a linked list of pending events that have not yet been recorded by the scheduler.

Event structures are placed in a corresponding shared memory region, accessible from the kernel regardless of which protection domain is currently active. Thus, when an upcall is performed, the event structures for the closest common scheduler in the hierarchy to the currently running thread and the upcall thread are efficiently located, and the urgency values in these structures are compared. If the upcall's brand has a lower numeric urgency field than the current thread, the upcall is immediately executed. The scenario is more complicated for the case when the upcall is completed. In this case, the kernel checks to see if a scheduler with permissions to schedule the previously executing thread has changed its preference for which thread to run. If this happens it will be reflected via the shared memory region between the scheduler and the kernel. Changes to the scheduling order might be due to the fact that the upcall invoked a scheduler to wake up a previously blocked thread. Additionally the kernel considers if another upcall was made while the current upcall was executing, but is deferred execution. If either of these conditions are true, then an upcall is made into the root scheduler allowing it to make a precise scheduling decision. However, the system is designed around the premise that neither of these cases occur frequently, and most often needs only to switch immediately back to the previous thread. This is typically the case with short running upcalls. However, if the upcalls execute for a more significant amount of time and the root scheduler is invoked, the consequent scheduling overhead is amortized.

Given these mechanisms which allow user-level component schedulers to communicate with the kernel, Composite supports low asynchronous event response times while still maintaining the configurability of scheduling policies at user-level.

### 2.3 Thread Accountability

Previous research has addressed the problem of accurately accounting for interrupt execution costs and identifying the corresponding thread or process associated with such interrupts [9, 34]. This is an important factor in real-time and embedded systems, where the execution time of interrupts needs to be factored into task execution times. Composite provides accurate accounting of the costs of asynchronous event notifications and charges them, accordingly, to corresponding threads.

As stated earlier, brands and upcalls enable efficient asynchronous notifications to be used by the system to deliver events, e.g. interrupts, without the overhead of explicit invocation of user-level schedulers. However, because thread switches can happen without direct scheduler execution, it is more difficult for the schedulers themselves to track total execution time of upcalls. If the problem were not correctly addressed, then the execution of upcalls might be charged to whatever thread was running when the upcall was initiated. We, therefore, expand the event structure within the shared kernel/scheduler region to include a counter, measuring progress of that event structure's associated thread. In our prototype implementation on the x86 architecture, we use the time-stamp counter (TSC) to measure the amount of time each thread spends executing by taking a reading whenever threads are switched. The previous reading is subtracted from the current value, to produce the elapsed execution time of the thread being switched out. This value is added to the progress counter in that thread's

event structure for each of its schedulers. On architectures without efficient access to a cycle counter, execution time can be sampled, or a simple count of the number of times upcalls are executed can be reported.

Observe that Composite provides library routines for common thread and scheduling operations. These ease development as they hide event structure manipulation and automatically update thread accountability information.

### 2.4 Efficient Scheduler Synchronization

When schedulers are implemented in the kernel, it is common to disable interrupts for short amounts of time to ensure that processing in a critical section will not be preempted. This approach has been applied to user-level scheduling in at least one research project [7]. However, given our design requirements for a system that is both dependable and predictable, this approach is not feasible. Allowing schedulers to disable interrupts could significantly impact response time latencies. Moreover, scheduling policies written by untrusted users may have faulty or malicious behavior, leading to unbounded execution (e.g., infinite loops) if interrupts are disabled. CPU protection needs to be maintained as part of a dependable and predictable system design.

An alternative to disabling interrupts is to provide a user-level API to kernel-provided locks, or semaphores. This approach is both complicated and inefficient, especially in the case of blocking locks and semaphores. As blocking is not a kernel-level operation in Composite, and is instead performed at user-level, an upcall would have to be performed. However, it is likely that synchronization would be required around wait queue structures, thus producing a circular dependency between kernel locks and the user scheduler, potentially leading to deadlocks or starvation. Additionally, it is unclear how strategies to avoid priority inversion could be included in such a scheme.

Preemptive non-blocking algorithms also exist, that do not necessarily require kernel invocations. These algorithms include both lock-free and wait-free variants [15]. Wait-free algorithms are typically more processor intensive, while lock-free algorithms do not necessarily protect against starvation. However, by judicious use of scheduling, lock-free algorithms have been shown to be suitable in a hard-real-time system [2]. It has also been reported that in practical systems using lock-free algorithms, synchronization delays are short and bounded [15, 17].

To provide scheduler synchronization that will maintain low scheduler run-times, we optimize for the common case when there is no contention, such that the critical section is not challenged by an alternative thread. We use lock-free synchronization on a value stored in the shared scheduler region, to identify if a critical section has been entered, and by whom. Should contention occur, the system provides a set of synchronization flags that are passed to the `cos_switch_thread` syscall, to provide a form of wait-free synchronization. In essence, the thread, $\tau_i$ waiting to access a shared resource "helps" the thread, $\tau_j$, that currently has exclusive access to that resource, by allowing $\tau_j$ to complete its critical section. At this point, $\tau_j$ immediately switches back to $\tau_i$. The assumption here is that the most recent thread to attempt entry into the critical section has the highest priority, thus it is valid to immediately switch back to it without invoking a scheduler. This semantic behavior exists in a scheduler library in Composite, so if it is inappropriate for a given scheduler, it can be trivially overridden. As threads never block when attempting access to critical sections, we avoid having to put blocking semantics into the kernel. The design decision to avoid expensive kernel invocations in the uncontested case is, in many ways, inspired by futexes in Linux [12].

Generally, many of the algorithms for non-blocking synchronization require the use of hardware atomic instructions. Unfortunately, on many processors the overheads of such instructions are significant due to factors such as memory bus locking. We have found that using hardware-provided atomic instructions for many of the common scheduling operations in Composite often leads to scheduling decisions having significant latencies. For example, both the kernel and user-level schedulers require access to event structures, to update the states of upcalls and accountability information, and to post new events. These event structures are provided on a per-CPU basis, and our design goal is to provide a synchronization solution that does not unnecessarily hinder thread execution on CPUs that are not contending for shared resources. Consequently, we use a mechanism called *restartable atomic sequences* (RASes), that was first proposed by Bershad [6], and involves each component registering a list of desired atomic assembly sections. These assembly sections either run to completion without preemption, or are restarted by ensuring the CPU instruction pointer (i.e., program counter) is returned to the beginning of the section, when they are interrupted.

Essentially, RASes are crafted to resemble atomic instructions such as compare and swap, or other such functions that control access to critical sections. Common operations are provided to components via Composite library routines [1]. The Composite system ensures that if a thread is preempted while processing in one of these atomic sections, the instruction pointer is *rolled back* to the beginning of the section, similar to an aborted transaction. Thus, when an interrupt arrives in the system, the instruction pointer of the currently executing thread is inspected and compared

[1] In this paper, we discuss the use of RASes to emulate atomic instructions but we have also crafted specialized RASes for manipulating event structures.

with the assembly section locations for its current component. If necessary, the instruction pointer of the interrupted thread is reset to the beginning of the section it was executing. This operation performed at interrupt time and is made efficient by aligning the list of assembly sections on cache lines. We limit the number of atomic sections per-component to 4 to bound processing time. The performance benefit of this technique is covered in Section 3.1.

```
cos_atomic_cmpxchg:
   movl %eax, %edx
   cmpl (%ebx), %eax
   jne cos_atomic_cmpxchg_end
   movl %ecx, %edx
   movl %ecx, (%ebx)
cos_atomic_cmpxchg_end:
   ret
```

**Figure 4. Example compare and exchange atomic restartable sequence.**

Figure 4 demonstrates a simple atomic section that mimics the `cmpxchg` instruction in x86. Libraries in Composite provide the `cos_cmpxchg(void *memory, long anticipated, long new_val)` function which expects the address in memory we wish to change, the anticipated current contents of that memory address, and the new value we wish to change that memory location to. If the anticipated value matches the value in memory, the memory is set to the new value which is returned, otherwise the anticipated value is returned. The library function calls the atomic section in Figure 4 with register `eax` equal to anticipated, `ebx` equal to the memory address, `ecx` equal to the new value, and returns the appropriate value in `edx`.

Observe that RASes do not provide atomicity on multiprocessors. To tackle this problem, however, either requires the use of true atomic instructions or the partitioning of data structures across CPUs. Note that in Composite, scheduling queue and event structures are easily partitioned into CPU-specific sub-structures, so our synchronization techniques are applicable to multi-processor platforms.

# 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 2.6.22 as the host operating system with a clock-tick (or *jiffy*) set to 10 milliseconds. Composite is loaded using the techniques from Hijack [21], and uses the networking device and timer subsystem of the Linux kernel, overriding all other control flow.

## 3.1 Microbenchmarks

Here we report a variety of microbenchmarks: (1) Hardware measurements for lower bounds on performance. (2) The performance of Linux primitives, as a comparison case. (3) The performance of Composite operating system primitives. All measurements were averaged over 100000 iterations in each case.

| Operation | Cost in CPU cycles |
|---|---|
| **User → kernel round-trip** | 166 |
| **Two user → kernel round-trips** | 312 |
| **RPC between two address spaces** | 1110 |

**Table 2. Hardware measurements.**

Table 2 presents the overheads we obtained by performing a number of hardware operations with a minimum number of assembly instructions specially tailored to the measurement. The overhead of switching between user-level to the kernel and back (as in a system call) is 166 cycles. Performing two of these operations approximately doubled the cost. Switching between two protection domains (page-tables), in conjunction with the two system calls, simulates RPC between components in two address spaces. It is notable that this operation on Pentium 4 processors incurs significant overhead.

| Operation | Cost in CPU cycles |
|---|---|
| **Null system call** | 502 |
| **Thread switch in same process** | 1903 |
| **RPC between 2 processes using pipes** | 15367 |
| **Send and return signal to current thread** | 4377 |
| **Uncontended lock/release using Futex** | 411 |

**Table 3. Linux measurements.**

Table 3 presents specific Linux operations. In the past, the `getpid` system call has been popular for measuring null system call overhead. However, on modern Linux systems, such a function does not result in kernel execution. To measure system-call overhead, then, we use `gettimeofday(NULL, NULL)`, the fastest system call we found. To measure context switching times, We use the NPTL 2.5 threading library. To measure context switch overhead, we switch from one highest priority thread to the other in the same address space using `sched_yield`. To measure the cost of IPC in Linux (an OS that is not specifically structured for IPC), we passed one byte between two threads in separate address spaces using pipes. To understand how expensive it is to create an asynchronous event in Linux, we generate a signal which a thread sends to itself. The signal handler is empty, and we record how long it takes to return to the flow of control sending the signal. Lastly, we measure the uncontended cost of taking and releasing a `pthread_mutex` which uses Futexes [12]. Futexes avoid invoking the kernel, but use atomic instructions.

| Operation | Cost in CPU cycles |
|---|---|
| **RPC between components** | 1629 |
| **Kernel thread switch overhead** | 529 |
| **Thread switch w/ scheduler overhead** | 688 |
| **Thread switch w/ scheduler and accounting overhead** | 976 |
| **Brand made, upcall not immediately executed** | 391 |
| **Brand made, upcall immediately executed** | 3442 |
| **Upcall dispatch latency** | 1768 |
| **Upcall terminates and executes a pending event** | 804 |
| **Upcall immediately executed w/ scheduler invocations** | 9410 |
| **Upcall dispatch latency w/ scheduler invocation** | 5468 |
| **Uncontended scheduler lock/release** | 26 |

**Table 4. Composite measurements.**

A fundamental communication primitive in Composite is a synchronous invocation between components. Currently, this operation is of comparable efficiency to other systems with a focus on IPC efficiency such as L4 [30]. We believe that optimizing the fast-path in Composite by writing it in assembly can further reduce latency. Certainly, the performance in Composite is an order of magnitude faster than RPC in Linux (as shown in Table 4).

As scheduling in Composite is done at user-level to ease customization and increase reliability, it is imperative that the primitive operation of switching between threads is not prohibitive. The kernel overhead of thread switching when accounting information is not recorded by the kernel is 0.22 microseconds. This is the lower bound for scheduling efficiency in Composite. If an actual fixed-priority scheduler is used to switch between threads which includes manipulating run-queues, taking and releasing the scheduler lock, and parsing event structures, the overhead is increased to 0.28 microseconds. Further, if the kernel maintains accounting information regarding thread run-times, and passes this information to the schedulers, overhead increases to 0.40 microseconds. The actual assembly instruction to read the time-stamp counter (`rdtsc`) contributes 80 cycles to the overhead, while locating and updating event structures provides the rest. We found that enabling kernel accounting made programming user-schedulers significantly easier. Even in this form, the thread switch latency is comparable to user-level threading packages that do not need to invoke the kernel, as reported in previous research [33], and is almost a factor of two faster than in Linux.

The overhead and latency of event notifications in the form of brands and upcalls is important when considering the execution of interrupt triggered events. Here we measure overheads of upcalls made under different conditions. First, when an upcall is attempted, but its urgency is not greater than the current thread, or if there are no inactive upcalls, the overhead is 0.16 microseconds. Second, when an upcall occurs with greater urgency than the current thread, the cost is 1.43 microseconds (assuming the upcall immediately returns). This includes switching threads twice, two user $\rightarrow$ kernel round-trips, and two protection domain switches. The time to begin executing an upcall, which acts as a lower-bound on event dispatch latency, is 0.73 microseconds. This is less than a thread switch in the same process in Linux. Third, when an upcall finishes, and there is a pending event, it immediately executes as a new upcall. This operation takes .33 microseconds.

A feature of Composite is the avoidance of scheduler invocations before and after every upcall. Calling the scheduler both before and after an upcall (that immediately returns) is 3.92 microseconds. By comparison, avoiding scheduler invocations, using Composite event structures, reduces the cost to 1.43 microseconds. The dispatch latency of the upcall is 2.27 microseconds when the scheduler is invoked, whereas it reduces to 0.73 microseconds using the shared event structures. It is clear that utilizing shared communication regions between the kernel and the schedulers yields a significant performance improvement.

Lastly, we compare the cost of the synchronization mechanism introduced in Section 2.4 against futexes. In Composite, this operation is barely the cost of two function calls, or 26 cycles, compared to 411 cycles with futexes. An illustration of the importance of this difference is that the cost of switching threads, which includes taking and releasing the scheduler lock, would increase in cost by 42% if futexes were used. The additional cost would rise as more event structures are processed using atomic instructions. As thread switch costs bound the ability of the system to realistically allow user-level scheduling, the cost savings is significant.

### 3.2 Case Study: Predictable Interrupt Scheduling

In the experiments in this section, we use network packet arrivals as our source of interrupts, and demultiplex the interrupts based on packet contents [18]. The demultiplexing operation is performed predictably, with a fixed overhead, by carefully choosing the packet parsing method [20].

To demonstrate the configurability of the Composite scheduling mechanisms, we implement a variety of interrupt management and scheduling schemes, and contrast their behavior. A component graph similar to that shown is Figure 1 is used throughout our experiments. In this figure, all shaded components are implemented in the kernel. The scheduling hierarchies under comparison are shown in Figure 5. Italic nodes in the trees are schedulers: *HW* is the hardware scheduler giving priority to interrupts, *FP_RR* is a fixed priority round-robin scheduler, and *DS* is a deferrable server with a given execution time and period. All such configurations include some execution at interrupt time labeled the *level 0 interrupt* handling. This is the interrupt execution that occurs before the upcall executes, and in our specific case involves network driver execution. The children below a scheduler are ordered from top to bottom, from higher to

lower priority. Additionally, dotted lines signify dependencies between execution entities in the system. The timer interrupt is not depicted, but the *FP_RR* is dependent on it. Task 3 is simply a CPU-bound background task.

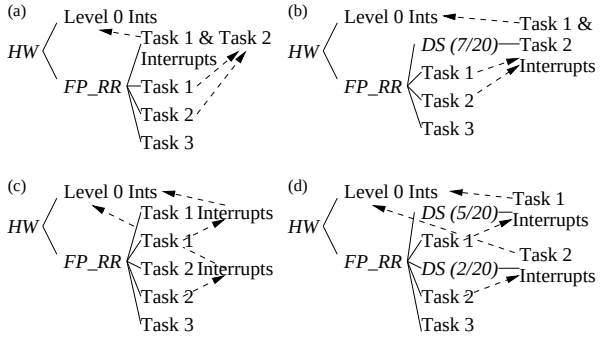


**Figure 5. Scheduling hierarchies implemented in Composite.**

Figure 5(a) depicts a system in which all interrupt handling is executed with the highest priority. Ignoring ad-hoc mechanisms for deferring interrupts given overload (such as `softirqd` in Linux), this hierarchy models the default Linux behavior. Figure 5(b) depicts a system whereby the processing of the interrupts is still done at highest priority, but is constrained by a deferrable server [31]. The use of a deferrable server allows for fixed priority schedulability analysis to be performed, but does not differentiate between interrupts destined for different tasks. Figure 5(c) depicts a system whereby the interrupts are demultiplexed into threads of different priority depending on the priority of the normal threads that depend on them. This ensures that high priority tasks and their interrupts will be serviced before the lower-priority tasks and their interrupts, encouraging behavior more in line with the fixed priority discipline. The interrupts are processed with higher priority than the tasks, as minimizing interrupt response time is often useful (e.g., to compute accurate TCP round-trip-times, and to ensure that the buffers of the networking card do not overflow, possibly dropping packets for the higher-priority task). Figure 5(d) depicts a system where interrupts for each task are assigned different priorities (and, correspondingly, brands). Each such interrupt is handled in the context of an upcall, scheduled as a deferrable server. These deferrable servers not only allow the system to be analyzed in terms of their schedulability, but also prevent interrupts for the corresponding tasks from causing livelock [19].

Streams of packets are sent to a target system from two remote machines, via Gigabit Ethernet. The packets arrive at the host, triggering interrupts, which execute through the device driver, and are then handed off to the Composite system. Here, a demultiplexing component in the kernel maps the execution to the appropriate upcall thread. From that point on, execution of the interrupt is conducted in a networking component in Composite, to perform packet processing. This takes 14000 cycles (a value taken from measurements of Linux network bottom halves [13]). When this processing has completed, a notification of packet arrival is placed into a mailbox, waking an application task if one is waiting. The tasks pull packets out of the mailbox queues, and processes them for 30000 cycles.

Figure 6(a) depicts the system when the interrupts have the highest priority (NB: lower numerical priority values equate to higher priority, or greater precedence). Packets arriving at sufficient rate cause livelock on the application tasks. The behavior of the system is not consistent or predictable across different interrupt loads. Figure 6(b) shows the system configured where the interrupts are branded onto a thread of higher precedence than the corresponding task requesting I/O. In this case, there is more isolation between tasks as the interrupts for Task 2 do not have as much impact on Task 1. Task 1 processes more packets at its peak and performs useful work for longer. Regardless, as the number of received packets increases for each stream, livelock still occurs preventing task and system progress.

Figure 6(c) depicts a system that utilizes a deferrable server to execute all interrupts. Here, the deferrable server is chosen to receive 7 out of 20 quanta. These numbers are derived from the relative costs of interrupt to task processing, leading to a situation in which the system is marginally overloaded. An analysis of a system with real-time or QoS constraints could derive appropriate rate-limits in a comparable fashion. In this graph, the interrupts for both tasks share the same deferrable server and packet queue. Given that half of the packets in the queue are from each task, it follows that even though the system wishes one task to have preference (Task 1), they both process equal amounts of packets. Though there is no notion of differentiated service based on Task priorities, the system is able to avoid livelock, and thus process packets across a wide variety of packet arrival rates.

Figure 6(d) differentiates interrupts executing for the different tasks by their priority, and also processes the interrupts on two separate deferrable servers. This enables interrupts to be handled in a fixed priority framework, in a manner that bounds their interference rate on other tasks. Here, the high priority task consistently processes more packets than the lower-priority task, and the deferrable servers guarantee that the tasks are isolated from livelock. The cumulative packets processed for Task 1 and 2, and the total of both are plotted in Figure 7(c). Both approaches that prevent livelock, by using deferrable servers, maintain high packets processing throughput. However, the differentiated service approach is the only one that both achieves high throughput, and predictable packet processing (with a 5 to 2 ratio for Tasks 1 and 2).

Figure 7 further investigates the behaviors of the two approaches using deferrable servers. Specifically, we wish to

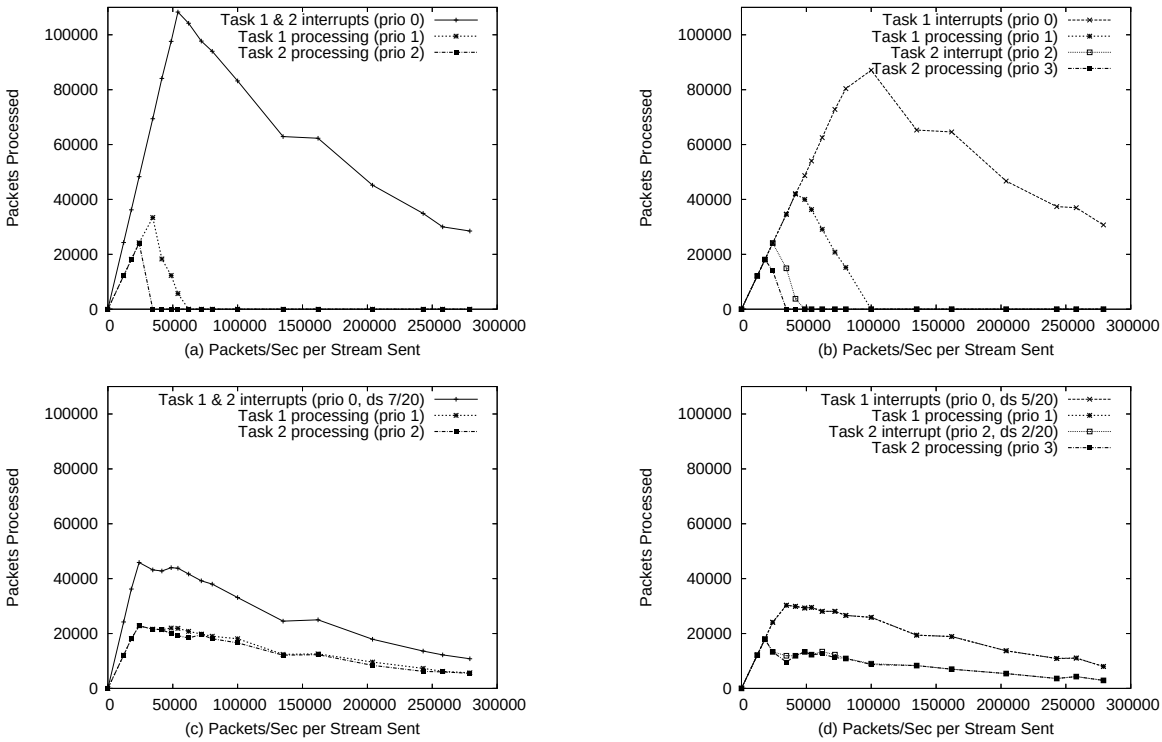


**Figure 6. Packets processed for two streams and two system tasks.**

study the ability of the system to maintain predictably differentiated service between the two tasks, given varying interrupt arrival rates. In this case, Task 1 is sent a constant stream of 24100 packets per second in Figure 7(a) and (b), and 488000 in Figure 7(d) and (e). The amount of packets per second sent to Task 2 varies along the x-axis. The results for both receive rates demonstrate that when all interrupts share the same deferrable server, allocation of processed packets to tasks is mainly dependent on the ratio of packets sent to Task 1 and Task 2. Separating interrupt processing into two different deferrable servers, on the other hand, enables the system to differentiate service between tasks, according to QoS requirements.

Figure 7(f) plots the total amounts of packets processed for the tasks in the system under the different hierarchies and constant packet receive rates. The differentiated service approach maintains a predictable allocation of processing time to the tasks consistent with their relative deferrable server settings. Using only a single deferrable server and therefore ignoring the task dependencies on interrupts, yields processing time allocations that are heavily skewed towards the task of lesser importance when it has more packets arrivals.

## 4 Related Work

Past research has put forth mechanisms to implement hierarchically structured user-level schedulers [11, 30]. Additionally, others have made the argument that user-level scheduling is useful for real-time systems, and have provided methods accommodating it in a middleware setting [4]. None of these works attempt to remove all notions of blocking and scheduling from the kernel. Additionally, these approaches, do not provide a mechanism for scheduling and accounting asynchronous events (e.g., interrupts) without recourse to costly scheduler invocations. We use this feature to achieve significantly lower event response times required for a predictable system, essentially by capturing scheduling semantics in event structures shared between user-space and the kernel.

The early demultiplexing of events [32, 18], and assigning interrupt execution to higher-level thread contexts [8, 9, 34] have been studied before. However, our approach of constructing a hierarchy of user-level schedulers for both interrupt and conventional thread scheduling in a component-based framework is novel. Our approach supports the construction of separate policies within each component in the hierarchy, along with corresponding isolation between components and the kernel itself. Thus, we offer a solution to the design of application-specific service policies in a low-cost, dependable and predictable (extensible) system.

Significant effort has been made to analytically study the compositional feasibility of constructing specific scheduling hierarchies [28, 24]. Others have investigated how to map abstract application QoS specifications into component schedulers to provide service guarantees [14]. Both of these

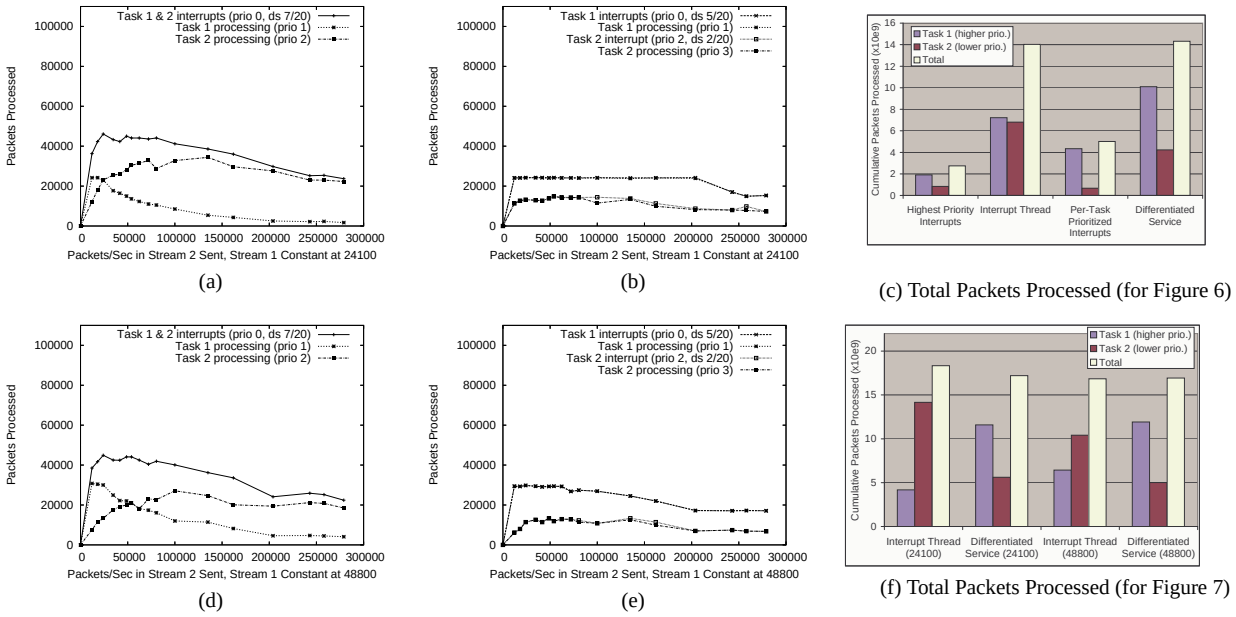


**Figure 7. Packets processed for two streams, one with constant rate.**

techniques are complementary to our work, and can be used to refine or verify a given hierarchy of user-level schedulers. Application-specific scheduling methods that take into account application constraints, and alter behavior based on system dynamics [23, 25] promise to lessen the semantic gap by offering a tighter coupling between application and scheduler.

## 5 Conclusions and Future Work

This paper presents the design of user-level scheduling hierarchies in the Composite component-based system. By providing support for user-defined component services that are separated from the kernel, untrusted or malicious software is prevented from jeopardizing the kernel. Moreover, component services themselves may be isolated from one another, thereby avoiding potentially adverse interactions. Collectively, this arrangement serves to provide a system framework that is both extensible and dependable. However, to ensure sufficient predictability for use in real-time domains, Composite features a series of low-overhead mechanisms, having bounded costs that are on par or better than competing systems such as Linux. Microbenchmarks show that Composite incurs low overhead in its various mechanisms to communicate between and schedule component services.

We describe a novel method of branding upcall execution to higher-level thread contexts. We also discuss the Composite approach to avoid direct scheduler invocation while still allowing full user-level control of scheduling decisions. Additionally, a lightweight technique to implement non-blocking synchronization at user-level, essential for the manipulation of scheduling queues, is also described. This is similar to futexes but does not require atomic instructions, instead relying on "restartable atomic sequences".

We demonstrate the effectiveness of these techniques by implementing different scheduling hierarchies, featuring various alternative policies, and show that it is possible to implement differentiated service guarantees. Experiments show that by using separate deferrable servers to handle and account for interrupts, a system is able to behave according to specific service constraints, without suffering livelock.

Future work includes porting more sophisticated scheduling policies to Composite. In doing so, we wish to provide a significant library of policies which applications and systems can compose into hierarchies as they see fit. Additionally, we will consider incorporating frameworks for more easily writing schedulers [5]. NB: Composite source code is available upon request.

## References


[1] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *USENIX Summer Symposium*, pages 93–113, 1986.

[2] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Trans. Comput. Syst.*, 15(2):134–165, 1997.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM symposium on Operating systems principles*, pages 95–109, 1991.

[4] T. Aswathanarayana, D. Niehaus, V. Subramonian, and C. Gill. Design and performance of configurable endsystem

scheduling mechanisms. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 32–43, 2005.

[5] L. P. Barreto and G. Muller. Bossa: a language-based approach to the design of real-time schedulers. In *The 10th International Conference on Real-Time Systems*, pages 19–31, Paris, France, March 2002.

[6] B. N. Bershad, D. D. Redell, and J. R. Ellis. Fast mutual exclusion for uniprocessors. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–233. ACM, 1992.

[7] J. Bruno, J. Brustoloni, E. Gabber, A. Silberschatz, and C. Small. Pebble: A component-based operating system for embedded applications. In *Proc. USENIX Workshop on Embedded Systems*, pages 55–65, 1999.

[8] L. E. L. del Foyo, P. Mejia-Alvarez, and D. de Niz. Predictable interrupt management for real time kernels over conventional PCx hardware. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 14–23, 2006.

[9] P. Druschel and G. Banga. Lazy receiver processing (LRP): a network subsystem architecture for server systems. In *Proceedings of the 2nd USENIX symposium on Operating Systems Design and Implementation*, pages 261–275, 1996.

[10] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the Winter USENIX Technical Conference and Exhibition*, pages 97–114, 1994.

[11] B. Ford and S. Susarla. CPU inheritance scheduling. In *Proceedings of the second USENIX Symposium on Operating Systems Design and Implementation*, pages 91–105, 1996.

[12] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Ottawa Linux Symposium*, 2002.

[13] G. Fry and R. West. On the integration of real-time asynchronous event handling mechanisms with existing operating system services. In *Proceedings of the International Conference on Embedded Systems & Applications*, pages 83–90, June 2007.

[14] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.

[15] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 217–230, 2001.

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

[17] F. Mehnert, M. Hohmuth, and H. Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *In Proc. of the 23rd IEEE Real-Time Systems Symposium*, December 2002.

[18] J. Mogul, R. Rashid, and M. Accetta. The packer filter: an efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating systems principles*, pages 39–51, 1987.

[19] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, 1997.

[20] R. Pagh and F. F. Rodler. Cuckoo hashing. *Lecture Notes in Computer Science*, 2161:121–144, 2001.

[21] G. Parmer and R. West. Hijack: Taking control of cots systems for real-time user-level services. In *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2007.

[22] 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*, 2007.

[23] A. Patil and N. Audsley. Implementing application specific RTOS policies using reflection. In *Proceedings of the IEEE Real-Time and Embedded Systems and Applications Symposium*, pages 438–447, 2005.

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

[25] S. Ruocco. User-level fine-grained adaptive real-time scheduling via temporal reflection. In *Proceedings of 27th IEEE International Real-Time Systems Symposium*, pages 246–256, 2006.

[26] S. Ruocco. A real-time programmer's tour of general-purpose L4 microkernels. In *EURASIP Journal on Embedded Systems*, 2008.

[27] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[28] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th IEEE Real Time Systems Symposium*, 2003.

[29] U. Steinberg, J. Wolter, and H. Hartig. Fast component interaction for real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 89–97, Washington, DC, USA, 2005.

[30] J. Stoess. Towards effective user-controlled scheduling for microkernel-based systems. *SIGOPS Oper. Syst. Rev.*, 41(4):59–68, 2007.

[31] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, 1995.

[32] D. Tennenhouse. Layered multiplexing considered harmful. In *Protocols for High-Speed Networks*, pages 143–148, North Holland, Amsterdam, 1989.

[33] R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: Scalable threads for Internet services. In *The 19th ACM Symposium on Operating Systems Principles*, 2003.

[34] Y. Zhang and R. West. Process-aware interrupt scheduling and accounting. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 191–201, Washington, DC, USA, 2006. IEEE Computer Society.