Predictable and Configurable Component-Based Scheduling in the COMPOSITE OS

GABRIEL PARMER, The George Washington University
RICHARD WEST, Boston University

This article 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 enforces 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. 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 diminishing 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.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems*; D.4.1 [**Operating Systems**]: Process Management—*Scheduling, Synchronization*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms: Performance, Reliability

Additional Key Words and Phrases: Scheduling, component-based operating systems

ACM Reference Format:
Parmer, G. and West, W. 2013. Predictable and configurable component-based scheduling in the composite OS. ACM Trans. Embedd. Comput. Syst. 13, 1s, Article 32 (November 2013), 26 pages.
DOI: http://dx.doi.org/10.1145/2536747.2536754

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

Authors' addresses: G. Parmer, Computer Science Department, George Washington University, 801 22nd Street NW, Suite 720E, Washington, DC 20052; email: gparmer@gwu.edu. R. West, Computer Science Department, Boston University, 111 Cummington St., Boston, MA 02215.

DOI: http://dx.doi.org/10.1145/2536747.2536754

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 [Bruno et al. 1999; Parmer and West 2007b] and micro-kernels [Liedtke 1995; Accetta et al. 1986] 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 that the interaction between such services remains predictable and efficient.

Of particular importance in this article is the ability to extend and configure the scheduling policies of the system for specific applications and subsystems. There is an incentive for applications to aggressively specialize scheduling around their requirements: custom scheduling policies enable increased predictability [Parmer and West 2007a; Goyal et al. 1996; Ruocco 2006; RTLinux; Aswathanarayana et al. 2005], and increased efficiency [von Behren et al. 2003]. Embedded systems benefit from customizability as scheduling policies are chosen based on the goals of the system: the target utilization, the schedulability analysis properties and assumptions, and the temporal isolation requirements. These factors vary considerably even when comparing only Rate-Monotonic and Earliest-Deadline First scheduling [Buttazzo 2005], let alone more complicated policies. Open real-time systems in which hard real-time tasks share a processor with best-effort applications additionally motivate customized scheduling policies. The best-effort portion of the system might require a specific time-sharing policy optimized for throughput, while the hard real-time subsystem requires a scheduler that provides timing guarantees and isolation from the rest of the system. However, there is a tension between enabling configurable scheduling policies for multiple subsystems or applications, and system reliability. If untrusted applications could load possibly faulty scheduling code into the kernel, they could cause system failure, or could monopolize the processor for their own benefit at the cost of other applications and services.

This article focuses on the design of predictable and efficient user-level services in our COMPOSITE component-based system. We show how a hierarchy of component-based schedulers can be supported with our system design, enabling different applications and subsystems to customize their temporal behavior, while more privileged schedulers ensure temporal isolation. A component, in this context, is a collection of code and data that defines some functionality that is accessed by other components through a well-defined interface. Importantly, these components are encapsulated: the only way to harness their functionality is through the interface. In COMPOSITE, all significant system policies are implemented in separate components, including memory management, scheduling, networking, and synchronization. Each component executes in a separate protection domain at user level. In this article, we focus on protection domains that provide memory isolation via hardware page-tables. Page tables are used to limit the ranges of memory that are accessible by each component. One component scheduler cannot adversely effect another scheduler by inadvertently accessing its data-structures, and if, for instance, the scheduler for a best effort subsystem fails, it will not effect the hard real-time scheduler. 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 accordance with the priority 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 [Zhang and West 2006; Druschel and Banga 1996], but which is addressed in our component-based system. Specifically, interrupts are given precedence over software threads scheduled by the OS/kernel regardless of if they do processing for a low-priority, or high-priority thread. This relationship between system threads and interrupt processing creates a dependency. For example, a thread waiting on a service request, such as a blocking I/O operation, forms a dependency on the subsequent interrupt handling triggered by a response from the I/O device itself. In embedded systems, both computation and reaction must predictably occur in response to physical stimuli. As this will often involve cooperation between interrupt-execution, and multiple system tasks, a careful consideration of these dependencies is required to produce predictable responses.

While micro-kernels offer a means by which new services can be deployed at user level, interprocess 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 [Liedtke 1995]. In COMPOSITE, communication between components is carried out by thread migration [Ford and Lepreau 1994; Parmer 2010] 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 [Steinberg et al. 2005; Ruocco 2008]. The preemption of a thread in a component, and subsequent invocation of that component by another thread can yield multiple threads concurrently executing in that component.

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 τ_1 is executing scheduler policy code. It is preempted while holding a semaphore, S, that ensures mutually exclusive access for the scheduling runqueue. An upcall thread, τ_2 (e.g., for a timer interrupt), tries to access the runqueue, attempts to take S, and cannot proceed. At this point, the kernel needs to know which thread to schedule, but as it has no scheduling information, it must upcall into the user-level scheduler to find the next thread to execute. However, this will cause yet another attempt to acquire S. Thus, neither τ_1 nor τ_2 are able to make effective progress. In Section 2.4 we detail a solution that avoids blocking synchronization primitives by using a wait-free technique. 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" [Bershad et al. 1992] and inspired by futexes [Franke et al. 2002]. During typical execution, synchronization is ensured without relying on atomic instructions or kernel invocations. This utility is used both for interthread 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

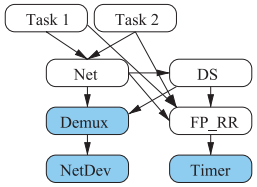

Fig. 1. An example component graph.

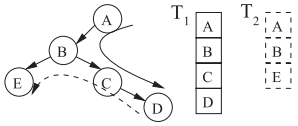

Fig. 2. Thread execution through components.

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

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. An invocation stack is maintained in each kernel thread structure. This stack is analogous to a thread's C runtime execution stack, but instead of tracking function invocations, it is a kernel data-structure that tracks component invocations. In addition to invoked components, each entry in the stack tracks the register state necessary to resume execution in a component that is returned to (for example, the instruction and stack pointers).

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 [Regehr and Stankovic 2001] whereby a series of successive 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

Table I. `cos_sched_cntl` Options

`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

subtree in the scheduling hierarchy. By comparison, scheduler activations [Anderson et al. 1991] 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. A root scheduler *promotes* other system components to be children schedulers. Processing time is allocated between the root's threads, and its children schedulers. The children schedulers themselves, can create children schedulers for whom they are the parent. In this way, CPU scheduling policy is delegated throughout a hierarchy of schedulers. A pointer within a component scheduler's kernel structure always refers to its parent scheduler. This is used to prevent the scheduling hierarchy from deviating from a tree structure. Cycles and even general acyclic graphs are prevented by ensuring that a component can never be promoted to a child scheduler if it already is a scheduler with a parent. This limitation ensures that parent schedulers can completely control and account for the CPU allocation of child subsystems. Likewise, parent schedulers can *demote* their children. This operation is recursive and removes the ability to schedule not only from direct children, but also their children (and so on). To bootstrap the system, the root scheduler automatically is given scheduling capabilities. Creating new child schedulers is done with feedback from abstract application requirements [Gai et al. 2001], or via application specified policies.

A component that has been promoted to scheduler status has access to the `cos_sched_cntl(operation, thd_id, other)` system call. The root scheduler automatically can use this system call. Here `operation` is simply a flag, the meaning of which is detailed in Table I, 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. This is necessary so that the root scheduler can define policy to prevent kernel memory denial of service attacks. For example, the root scheduler can implement thread creation policies (e.g., quotas) for specific subsystems or applications. A parent scheduler can allow a child scheduler to schedule a specific thread by *granting* scheduling privileges to it via `cos_sched_cntl(COS_SCHED_GRANT_THD,...)`. These privileges can only be granted on threads that the granting scheduler already is able to schedule. The root scheduler automatically can schedule all threads. Likewise, the `COS_SCHED_REVOKE_THD` flag can be used to recursively remove scheduling privileges from child schedulers. In conjunction, these calls enable a parent to strictly control the subsystems that can define temporal policy for specific threads. Importantly, the COMPOSITE kernel prevents a single thread from being schedulable by two children again so that a scheduler can maintain strict control and accounting information for its threads.

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

Threads are both created and passed up to other components via the `thd_id_t cos_thd_cntl (component_id, flags, arg1, arg2)` syscall. To create a new thread, the root scheduler makes a system call with the `COS_THD_CREATE` flag. If a nonroot 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.

Section 2.2 will discuss mechanisms for efficiently and predictably scheduling interrupt execution and maintaining accurate processing time accounting for system threads. This support requires a shared region of memory between each scheduler and the kernel, and for specific threads to be associated with entries within this shared region. The shared region is setup for a scheduler by invoking `cos_sched_cntl(COS_SCHED_SHARED_REGION,...)` and passing in the address of the region. A specific thread that has been granted to a scheduler can be associated with a entry in that region using the `COS_SCHED_THD_EVT` flag. Only components that have been promoted to a scheduler can setup a shared region, and only if a scheduler has been granted scheduling abilities to a thread can it successfully use `COS_SCHED_THD_EVT`.

2.1. Implementing Component Schedulers

Unlike previous systems [Ford and Susarla 1996; Aswathanarayana et al. 2005; Stoess 2007] that provide user-level scheduling, 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. Importantly for real-time systems, the schedulers—in coordination with lock components—define the blocking and scheduling semantics for shared resource contention [Parmer and Song 2010]. This is in contrast to systems with user-level schedulers whereby the scheduler is informed of a thread's blocking, but is not given any context regarding why the thread is blocking [Ford and Susarla 1996; Aswathanarayana et al. 2005; Stoess 2007]. In this way, user-level components and schedulers have the power to incorporate appropriate protocols to combat priority inversion (e.g., priority inheritance or ceiling protocols [Sha et al. 1990]).

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 without compromising predictability or efficiency? 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. Additionally, if a thread blocks in wait for a specific event (i.e., an interrupt from an external stimulus), a scheduler should be able to ensure that event did indeed happen before the thread is woken. Schedulers of higher trust (parent schedulers) must be able to ensure that child schedulers with the ability to dispatch to a specific blocked threads will not interfere with the blocking semantics of that parent scheduler. Thus, to enable 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 currently 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. Brands are given priorities by schedulers corresponding to the 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

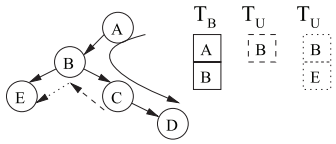

Fig. 3. Branding and upcall execution.

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.

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 reinstantiated.

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, implement 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 invocations of the schedulers 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 I.

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 a *priority* field. These priority fields are scheduler-specific and simply imply a total ordering of tasks. Depending on the policy of a given scheduler, these 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

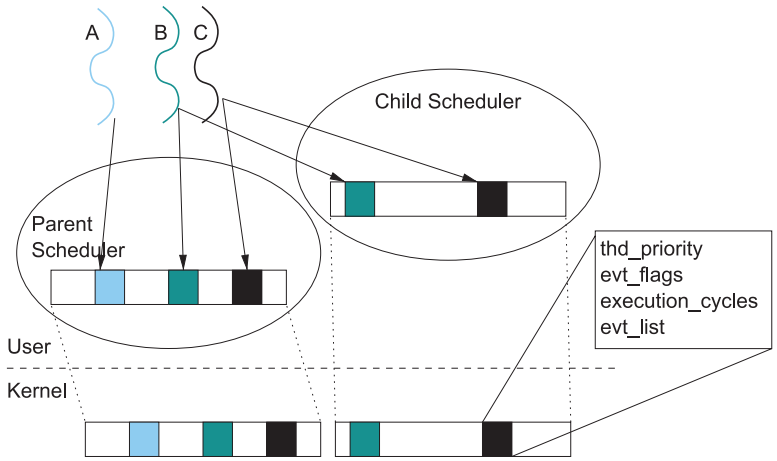

Fig. 4. Shared kernel-scheduler region used to 1) pass event status information (via the `evt_flags` field) and thread execution time from kernel to scheduler (via the `execution_cycles` field), and 2) pass information regarding relative thread (and upcall) priority from scheduler to kernel (via the `thd_priority`) field. Thread A is scheduled by the parent scheduler, while both B and C are schedulable by the parent and child scheduler (i.e., the parent granted scheduling privileges to the child).

priority field represent higher execution priority 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 processed 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. Figure 4 illustrates these data structures and their relationship to the schedulers, the kernel, and system threads. 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 priority values in these structures are compared. If the upcall's brand has a lower numeric priority field than the current thread, the upcall is immediately executed. The scenario is more complicated for the case when the upcall has completed execution, and another thread must be chosen for execution. 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: does the scheduler still give highest priority to the thread that was interrupted by the interrupt? 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 the system 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. This mechanism is tailored to avoid

```
thd_id_t next_thd;
do {
    process_evts();
    next_thd = scheduling_policy();
} while (cos_switch_thread(next_thd) != SUCCESS);
```

Fig. 5. Simplified scheduler component code. Kernel support prevents the scheduler from missing events that occur between when `process_evts` is called, and when the next thread is dispatched.

excessive overheads for I/O devices with a high interrupt frequency. In such cases, it is important to avoid switching to schedulers and incurring the associated overhead.

Kernel-Scheduler Synchronization. Interrupts can execute at any time, causing important changes to upcall state. The scheduler will typically process these events before making policy decisions, and then dispatching the next thread. This protocol is depicted in Figure 5. When processing events, threads might be added or removed from the run-queue as upcalls either activate or complete execution.

Asynchronous interrupts can occur in between when a thread has processed all pending events, and when a thread is dispatched. Therefore, any such events will not be considered when calculating the next thread for execution. This can lead to lower priority threads being executed when higher priority threads are runnable. To prevent this in COMPOSITE, a flag exists in the shared region of memory between each scheduler and kernel that the kernel sets to 1 when an event occurs. As part of the `process_evts` call, the user-level scheduler resets this flag to 0. Thus, when a thread calls `cos_switch_thread`, the kernel can check this flag. If it is set, then an event has occurred between the time the scheduler processed the events, and when it dispatched. In such a case, the kernel returns a specific error code (here denoted as `!SUCCESS`) to the scheduler that then processes the events and recomputes which thread to execute next.

This protocol enables user-level schedulers to maintain control over scheduling policy while still allowing events to occur asynchronously and with a low latency. The kernel ensures that consistent scheduling decisions are made by actively enforcing that events must be processed before valid scheduling decisions can be made.

Associating Interrupts with Brands. COMPOSITE provides the `brand_wire` system-call that associates a brand with a specific hardware interrupt source. When an interrupt arrives from that source, it automatically attempts to execute an upcall associated with that brand. This requires the normal checks to see if the brand has a higher priority than the currently executing thread, and only if it does will the upcall execute immediately.

It should be noted here that a limited amount of computation occurs at interrupt-time. That is to say that interrupts occur at hardware-defined priorities typically higher than the current software thread. We attempt to promote this execution to a schedulable upcall thread as soon as possible to minimize the impact of the interrupt's execution, and to schedule the majority of the interrupt-driven execution. Hardware techniques exist that can prevent even this small amount of interrupt-time processing. These include intelligent NICs that can classify incoming packets and prioritize interrupts [Druschel and Banga 1996], or unified hardware/software prioritization schemes [del Foyo et al. 2006]. Though these techniques could be adopted in COMPOSITE, we leave this for future work. Instead, this article investigates the complementary software support to determine how upcalls (whenever they occur) can be efficiently and predictably scheduled.

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 [Druschel and Banga 1996; Zhang and West 2006]. 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, for instance, 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 and is depicted in Figure 4 as `execution_cycles`. When a thread in the scheduler processes the events, it will update the execution times of each thread with an event an a corresponding execution cycles value. 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 [Bruno et al. 1999]. 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. Assuming blocking locks were provided by the kernel, the scheduler would use one to protect its shared data structures such as the run-queue. Before a scheduling policy decision were made, this lock would be obtained. However, this solution is untenable. If the kernel-provided lock were currently taken by another thread, the current thread attempting to take the lock would have to block. However, the action of blocking requires a scheduling policy to decide which thread to execute next. As the scheduling policy exists in a user-level component, kernel locks have to invoke the user-schedulers. Unfortunately, when the

user-component then attempts to access its run-queue, it will again find that the lock is contended causing this entire process to be repeated without forward progress. This is an inherent circular dependency between the user-scheduler and kernel-provided locks that will produce livelock and starvation, and thus, unpredictable behavior. Additionally, with kernel-provided locking, it is unclear how strategies to avoid priority inversion could be included in such a scheme as the kernel is oblivious to thread priorities. Due to these complications, the COMPOSITE kernel instead achieves mutual exclusion with nonblocking algorithms.

Preemptive nonblocking algorithms also exist, that do not necessarily require kernel invocations. These algorithms include both lock-free and wait-free variants [Hohmuth and Härtig 2001]. 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 [Anderson et al. 1997]. It has also been reported that in practical systems using lock-free algorithms, synchronization delays are short and bounded [Hohmuth and Härtig 2001; Mehnert et al. 2002].

To provide scheduler synchronization that will maintain low scheduler runtimes, 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, `COS_SCHED_SYNC_BLOCK` and `COS_SCHED_SYNC_UNBLOCK`, to provide a form of wait-free synchronization. In essence, the thread, τ_i waiting to access a shared resource is "helped" by thread τ_j, that currently has exclusive access to that resource, by allowing τ_j to complete its critical section, and immediately switch back to τ_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. These synchronization primitives are used to implement a critical section that protects a scheduler's data-structure. 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 [Franke et al. 2002].

Scheduler/Kernel Synchronization Details. Correct implementation of the `cos_switch_thread` is surprisingly difficult. Specifically, the manner in which the arguments to the system-call are passed has a great influence on the predictability of the call. If the identifier for the next thread to dispatch to is passed to the kernel via register, a condition can occur in which the incorrect thread (i.e., not the one with highest priority) is switched to leading to unpredictable periods of priority inversion. For example, a thread τ_0 executing scheduler policy to determine the next thread to execute will be holding the scheduler's critical section and, before calling `cos_switch_thread`, will release the critical section. Between the time when the critical section is released, and when the system call is made to dispatch to the next thread, threads could be woken up, or blocked. If the next thread is stored in thread-local location (i.e., registers or the thread's stack), then τ_0 would ask the kernel to dispatch to an incorrect thread—not the highest priority runnable thread. This situation is depicted in Figure 6(a), where the thread is preempted between releasing the critical section, and switching to the next thread.

In COMPOSITE, we ensure that this situation does not occur by preventing a thread from dispatching to another thread with this outdated state. Specifically, when a thread

```
do {
    thd_id_t next_thd;
    take_scheduler_crit_sect();
    next_thd = scheduling_policy();
    release_scheduler_crit_sect();
} while (cos_switch_thread(next_thd) != SUCCESS);
```

(a)

```
do {
     thd_id_t next_thd;
    take_scheduler_crit_sect();
    next_thd = scheduling_policy();
    shared_region.next_thread = next_thd;
    release_scheduler_crit_sect();
} while (cos_switch_thread(next_thd) != SUCCESS);
```

(b)

Fig. 6. (a) Thread-local variable used to dispatch to next thread. (b) A memory region shared by scheduler and the kernel is updated by all dispatching threads, and is used to avoid priority inversion.

chooses the next thread to execute given the scheduler's policy, this value is stored in the region of memory shared between kernel and the scheduler rather than in a register. This is depicted in Figure 6(b). Whenever any dispatching occurs, the memory region is updated to reflect this. The kernel implementation detects if `cos_switch_thread` is invoked with a stale thread identifier by comparing the next thread passed in the register with the next thread in the shared memory region. In the case of a discrepancy, the kernel returns an appropriate error code from the system call (not `SUCCESS`). The calling thread then queries the scheduling policy to recompute the proper next thread to execute.

Predictable, Efficient Atomic Instructions. 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 et al. [1992], 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 with the assembly

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

```
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
```

Fig. 7. Example compare and exchange atomic restartable sequence.

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. As this operation can only occur when an interrupt triggers, RASs have an additional benefit that they avoid the traditional need for either language-based, or architectural memory barriers that are required in many nonblocking algorithms. These barriers enforce ordering between loads and stores, and can impose significant overhead. This ordering is a natural side-effect of interrupts.

The roll-back to the beginning of a RAS is performed at interrupt time and is made efficient by aligning the per-component list of assembly sections on cache lines. We limit the number of atomic sections per-component to 4 to bound processing time.

The RAS mechanism is useful for multiple reasons: (1) it can provide atomic sequences even on embedded architectures that do not have support for atomic instructions, (2) it can significantly improve efficiency on architectures with inefficient atomic instruction support as it does not require memory barriers or special architectural support, (3) and it can lower system interference as it does not require bus or interconnect locking. RAS are used in COMPOSITE to process event structures, and to acquire execution stacks on component invocation. The performance benefit of this technique is covered in Section 3.1. The largest deficiency of RAS is that they do not provide synchronization between multiple processors. This is discussed shortly.

Figure 7 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 7 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`.

Synchronization and Multi-Processors. Observe that RASes do not provide atomicity on multi-processors. They do not utilize memory barriers, or other hardware support, so no memory access ordering is ensured by them across different processors. For a large class of data-structures that are partitioned across processors (e.g., COMPOSITE event structures, run-queues for many scheduling policies), they can be used without modification. Such partitioning is often used to ensure scalable data-structure access times with an increasing number of processors. However, for data structures that are shared between processors, traditional methods such as atomic instructions must instead be used for synchronization. Complicated synchronization techniques use a combination of both partitioned and global data-structures to optimize for certain types of data-structure access (i.e., read vs. write) [Gamsa et al. 1999].

One topic relevant to multi-processor systems is interprocessor interrupts (IPIs) that are used for notification or cooperation between processors. The basic mechanism of brands and upcalls is adapted in a straightforward manner to expose IPIs to

```
int thread_new(struct sched_thd *t);
int thread_remove(struct sched_thd *t);

int time_elapsed(struct sched_thd *t, u32_t processing_time);
int timer_tick(int num_ticks);

struct sched_thd *schedule(void);

int thread_block(struct sched_thd *t);
int thread_wakeup(struct sched_thd *t);
```

Fig. 8. Functions that different scheduling policies must implement.

components in COMPOSITE. A brand is associated with each IPI, and when an interrupt occurs, the execution of the brand's upcall is triggered.

Predictability of COMPOSITE *Synchronization Mechanisms.* It is not immediately obvious that the synchronization mechanisms used in COMPOSITE are predictable. The loops in Figures 5 and 6 do not appear to be bounded, and the number of times that a RAS can be rolled-back also does not seem bounded. Lock-free algorithms pose comparable challenges in that a thread performs some computation and at some point checks if it can "commit" its work. If another thread has interfered (by altering the data-structure), the computation must be repeated. The same techniques used to make lock-free algorithms predictable, thus statically analyzable, are used to perform analysis on COMPOSITE [Anderson et al. 1997]. The general theme of these techniques is to derive a maximum bound on the number of preemptions for each thread (e.g., which is related to period, WCET, and priority of different threads in a fixed priority periodic task model). Each of the sections of synchronization code that can be recomputed (loops and RAS) are only reexecuted due to preemptions. The contribution to a thread's WCET of the COMPOSITE synchronization mechanisms is the WCET of the content of each individual loop (from Figures 5 and 6) and the RASs times the maximum number of preemptions.

Scheduling policies could also ensure that RAS and scheduling loops are not unbounded. Dynamic priority policies can increase a thread's priority to decrease the number of times a thread is preempted to ensure forward progress.

2.5. Scheduler Implementation and Interfaces

The mechanisms for implementing user-level, component-based, customizable schedulers in COMPOSITE are relatively complex. It would be unrealistic to assume that scheduler developers have mastered these intricacies. Thus we abstract a scheduler's codebase into two sections: (1) the scheduler policy, and (2) a library that interfaces with the COMPOSITE kernel via the event structures in the shared scheduler/kernel memory region, performs synchronization around run-queues, and dispatches threads.

Scheduler Policy. Policy implementers must provide only a small set of functions that are called whenever an appropriate event has taken place. The prototypes of these functions are depicted in Figure 8. They include functions to create and remove threads (add and remove them from consideration for scheduling), time-management functions describing how long specific threads have executed, and how many timer ticks have elapsed (for child schedulers, multiple ticks might elapse before they execute), a function to decide the next thread to execute, and functions to block a thread and wake it (i.e., remove it from or add it to the run-queue, respectively).

Using this interface, a static priority, round-robin scheduler with a large amount of debugging functionality, comments, and white-space is implemented in 350 lines of C code.

```
void sched_schedule(void) {
    thd_id_t next_thd;
    do {
        take_scheduler_crit_sect();
        process_evts();
        // invoke the scheduling policy:
        next_thd = schedule();
        shared_region.next_thread = next_thd;
        release_scheduler_crit_sect();
    } while (cos_switch_thread(next_thd) != SUCCESS);
}
```

Fig. 9. Implementation of library code for scheduling.

```
int sched_block(void);
int sched_block_dep(thd_id_t dependency_thd);
int sched_wakeup(thd_id_t thd_id);

int sched_create_thread(component_id_t c_id, char *parameters);
void sched_exit(thd_id_t thd_id);

int sched_component_take(void);
int sched_component_release(void);

int sched_timeout_thd(void);
void sched_timeout(unsigned long amnt);
```

Fig. 10. The component interface for a scheduler.

Scheduler Library. Here we detail how all of the mechanisms of Section 2 are integrated, giving an example from the scheduler library. Additionally, we discuss the scheduler component's interface with other components.

Given the mechanisms described previously for synchronization, events, and accountability, and the functions provided by the scheduler policy, Figure 9 depicts the scheduler library code that decides the next thread to run, and switches to it. `process_evts`, in this case, has multiple duties as it parses all events that have happened since its last invocation and performs a number of operations: (1) remove or add threads as appropriate from the run-queue via `thread_block` and `thread_wakeup`, (2) update thread and upcall execution times via `time_elapsed`, and (3) clear the pending event flag (Section 2.4).

Figure 10 depicts the major functions constituting the scheduler component's interface that other components use to access the scheduler's functionality. These include functions to block a calling thread (perhaps because it is waiting for an event), to block a thread with a dependency (e.g., because it is contending a shared resource), and to wake up a thread that has previously blocked. Additionally, there are functions to create a new thread in a specific component, passing a set of parameters that might include, for instance, a static priority, and to discontinue execution of a thread. The rest of the functions in the interface provide higher-level abstractions to calling components. `sched_component_take` and `sched_component_release` provide a critical section abstraction for each calling component that is similar to the critical section that the scheduler uses to protect its own structures. Additionally, a component in the system can be designated to control time-outs. This component uses the last two functions to set a thread to be the "wake-up thread", and to set a timeout for that wake-up thread. The scheduler, then, simply wakes up that thread if the appropriate amount of clock ticks have elapsed. The wake-up thread returns to its component and can provide higher-level abstractions for timeouts for the rest of the system's threads. This interface, though simple, has been used to support complex applications such as a web-server [Parmer 2009]. Specialized scheduling policies can extend this interface as

Table II. Hardware Measurements

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

Table III. Linux Measurements

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

they see fit, for example to provide reflexive facilities [Patil and Audsley 2005; Ruocco 2006].

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 [Parmer and West 2007a], 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.

Table II 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 from 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.

Table III 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 high-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 (passing the byte from process A to B and back to A). 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.

Table IV. COMPOSITE Measurements

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

Last, we measure the uncontended cost of taking and releasing a `pthread_mutex` which uses Futexes [Franke et al. 2002]. Futexes avoid invoking the kernel, but use atomic instructions.

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 [Stoess 2007]. 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 IV).

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 [von Behren et al. 2003], 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 priority 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 priority 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. There is no direct comparison in Linux to upcall dispatch. Upcall dispatch certainly imposes an overhead over Linux ISR dispatch. However, if the hardware event must be processed by user-level, upcall cost is perhaps best compared to signal or thread

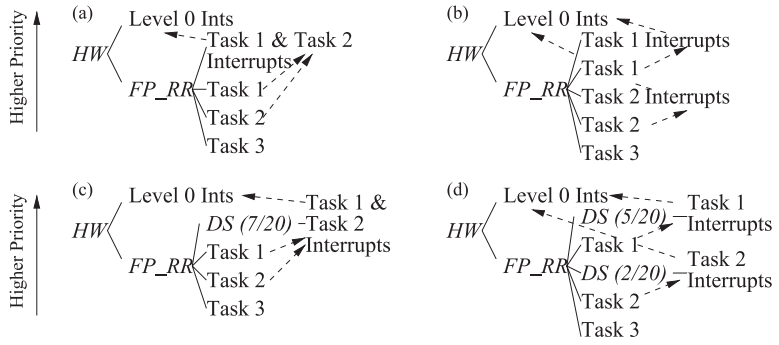

Fig. 11. Scheduling hierarchies implemented in COMPOSITE.

switch latencies to simulate the event being passed to an application thread to process. In both of these cases, upcall dispatch is competitive. Third, when an upcall finishes, and there is a pending event associated with its brand, 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.

Last, 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 [Mogul et al. 1987]. The demultiplexing operation is performed predictably, with a fixed overhead, by carefully choosing the packet parsing method [Pagh and Rodler 2001]. Clients send UDP packets with a payload of one byte. This study focuses on scheduling and reacting to interrupts, thus we use small packets to prevent interference from data copying.

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 11. 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. Dependencies arise when a task blocks and waiting for interrupt data, thus making their further execution dependent on the processing of interrupts. The timer interrupt is not depicted, but the *FP_RR* is dependent on it. Task 3 is simply a CPU-bound background task.

In this section, we investigate the ability of COMPOSITE to effectively and predictably schedule using customized, user-level policies, while accurately maintaining execution times of different threads. Additionally, we show how schedulers can be arranged in a hierarchy to change the temporal behavior of the system. As an example application, we demonstrate accurate thread and upcall execution by providing differentiated service for tasks processing networking packets. In this work we do not attempt to remove all level 0 interrupt processing (e.g., by using hardware support such as in [Druschel and Banga 1996]), thus there will be some interference from this execution even for high priority tasks. We leave it as future work to integrate such mechanisms into the abstractions provided by COMPOSITE. The effectiveness of COMPOSITE's mechanisms, then is measured in terms of if schedulers can effectively prevent interference between its schedulable entities (thread and upcalls).

Figure 11(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 11(b) 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 11(c) depicts a system whereby the processing of the interrupts is done at highest priority, but is constrained by a deferrable server [Strosnider et al. 1995]. 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 11(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 system schedulability analysis to be performed, but also diminish the livelock caused by interrupts for the corresponding tasks [Mogul and Ramakrishnan 1997].

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 [Fry and West 2007]). 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 12(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 a sufficient rate cause livelock on the application tasks. The behavior of the system is not consistent or predictable across different interrupt loads. Figure 12(b)

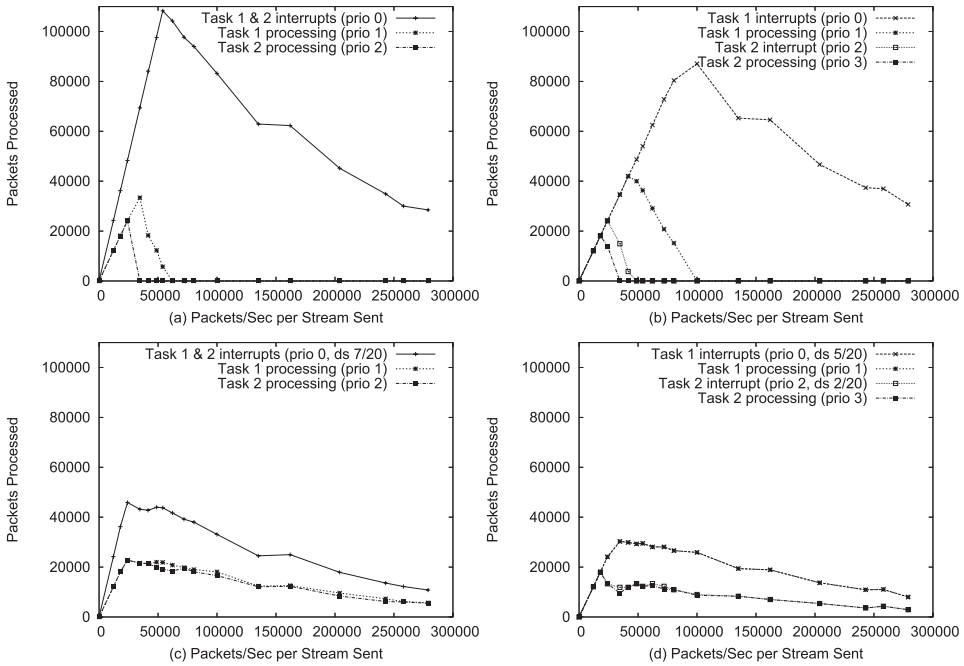

Fig. 12. Packets processed for two streams and two system tasks.

shows the system configured where the interrupts execute as upcalls associated with a brand with a higher priority than the corresponding dependent task, but where upcalls associated with Task 2 have a lower priority than Task 1 and its associated upcalls. In this case, there is less interference between tasks as the upcall execution for Task 2 does 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 at 100K packets per second preventing task and system progress.

Figure 12(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 at 250K packets per second. To avoid livelock independent of the packets per second, hardware support is required to decrease interrupt rate [Mogul and Ramakrishnan 1997; Druschel and Banga 1996]. Here we study if the COMPOSITE mechanisms can be used to enable component-based schedulers to effectively control software threads and upcalls, but results still demonstrate level 0 interrupt overheads that are not controlled by schedulers.

Figure 12(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

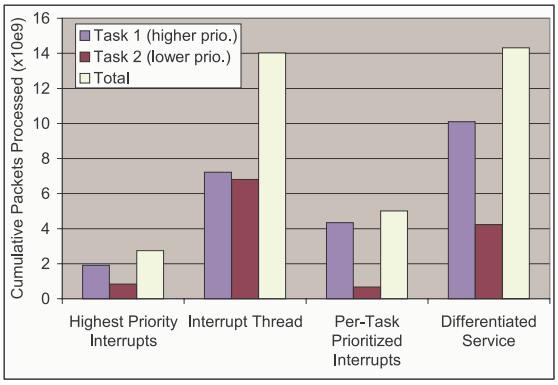

Fig. 13. Total packets processed for Figure 12.

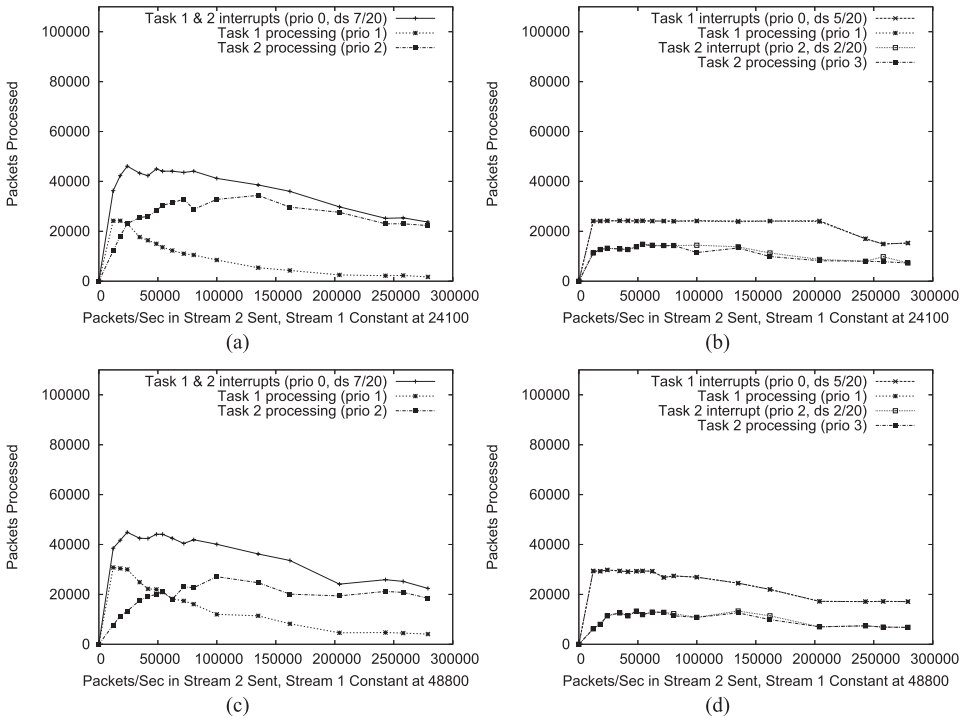

Fig. 14. Packets processed for two streams: Stream 1 has packets for Task 1, and Stream 2 has packets for Task 2.

their interference rate on other tasks. Here, the high priority task consistently processes more packets than the lower-priority task, and the deferrable servers diminish the interference of interrupt-driven execution. The cumulative packets processed for Task 1 and 2, and the total of both are plotted in Figure 13(c). Both approaches that use deferrable servers to diminish interrupt interference maintain high 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 14 further investigates the behaviors of the two approaches using deferrable servers. Specifically, we wish to study the ability of the system to maintain predictably

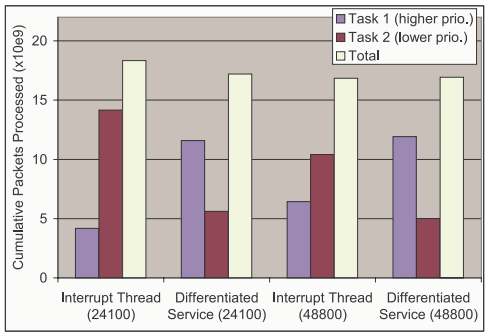

Fig. 15. Total packets processed for Figure 14.

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 14(a) and (b), and 488000 in Figure 14(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 15 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 priority when it has more packets arrivals.

4. RELATED WORK

Past research has put forth mechanisms to implement hierarchically structured user-level schedulers [Ford and Susarla 1996; Stoess 2007]. 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 [Aswathanarayana et al. 2005]. 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 [Tennenhouse 1989; Mogul et al. 1987], and assigning interrupt execution to higher-level thread contexts [del Foyo et al. 2006; Druschel and Banga 1996; Zhang and West 2006] have been studied before. We perform predictable upcall scheduling using a hierarchy of customizable, component-based schedulers. Each is in a separate protection domain, thus providing 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 [Shin and Lee 2003; Regehr and Stankovic 2001]. Others have investigated how to map abstract application QoS specifications into component schedulers to provide service guarantees [Gai et al. 2001]. Both of these

techniques are complementary to our work, and can be used to refine or verify a given hierarchy of user-level schedulers. Hierarchical scheduling can be used to enable child schedulers to be customized to their subsystems, while still providing temporal isolation between them via the parent scheduler. In contrast, resource kernels [Rajkumar et al. 2001] do not focus on enabling customizable scheduling policies, but do ensure temporal isolation between subsystems. Application-specific scheduling methods that take into account application constraints, and alter behavior based on system dynamics [Patil and Audsley 2005; Ruocco 2006] promise to lessen the semantic gap by offering a tighter coupling between application and scheduler.

This article is an extended version of Parmer and West [2008].

5. CONCLUSIONS AND FUTURE WORK

This article 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. These capabilities are essential for future cyber-physical systems that require novel temporal policies, and dependability in the face of failures. 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. It is difficult to achieve hard real-time execution on commodity hardware as there are many unpredictable and uncontrollable sources of interference (e.g., nonmaskable System Management Mode interrupts on x86, and shared interrupt lines). In as much as the underlying hardware can be controlled by the system, we have shown that scheduling policy can be efficiently and predictably implemented in components at user level. Additionally, a lightweight technique to implement nonblocking 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 [Barreto and Muller 2002]. Documentation, information, and code for Composite can be found at http://www.seas.gwu.edu/ gparmer/composite.html.

REFERENCES

ACCETTA, M. J., BARON, R. V., BOLOSKY, W. J., GOLUB, D. B., RASHID, R. F., TEVANIAN, A., AND YOUNG, M. 1986. Mach: A new kernel foundation for unix development. In *USENIX Summer Conference*. 93–113.

ANDERSON, J. H., RAMAMURTHY, S., AND JEFFAY, K. 1997. Real-time computing with lock-free shared objects. *ACM Trans. Comput. Syst. 15,* 2, 134–165.

ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. 1991. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP'91)*. ACM Press, New York, 95–109.

ASWATHANARAYANA, T., NIEHAUS, D., SUBRAMONIAN, V., AND GILL, C. 2005. Design and performance of configurable endsystem scheduling mechanisms. In *Proceedings of the 11th IEEE Real Time Symposium on Embedded Technology and Applications (RTAS'05)*. IEEE, 32–43.

BARRETO, L. P. AND MULLER, G. 2002. Bossa: a language-based approach to the design of real-time schedulers. In *Proceedings of the 10th International Conference on Real-Time Systems (RTS'02)*. 19–31.

BERSHAD, B. N., REDELL, D. D., AND ELLIS, J. R. 1992. Fast mutual exclusion for uniprocessors. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 223–233.

BRUNO, J., BRUSTOLONI, J., GABBER, E., SILBERSCHATZ, A., AND SMALL, C. 1999. Pebble: A component-based operating system for embedded applications. In *Proceedings of the. USENIX Workshop on Embedded Systems*. 55–65.

BUTTAZZO, G. C. 2005. Rate monotonic vs. edf: judgment day. *Real-Time Syst. 29,* 1, 5–26.

DEL FOYO, L. E. L., MEJIA-ALVAREZ, P., AND DE NIZ, D. 2006. Predictable interrupt management for real time kernels over conventional pc hardware. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*. IEEE, 14–23.

DRUSCHEL, P. AND BANGA, G. 1996. Lazy receiver processing (lrp): A network subsystem architecture for server systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*. ACM Press, New York, 261–275.

FORD, B. AND LEPREAU, J. 1994. Evolving mach 3.0 to a migrating thread model. In *Proceedings of the USENIX Technical Conference and Exhibition*. 97–114.

FORD, B. AND SUSARLA, S. 1996. Cpu inheritance scheduling. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*. ACM Press, New York, 91–105.

FRANKE, H., RUSSELL, R., AND KIRKWOOD, M. 2002. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Proceedings of the Ottawa Linux Symposium*.

FRY, G. AND WEST, R. 2007. On the integration of real-time asynchronous event handling mechanisms with existing operating systemservices. In *Proceedings of the International Conference on Embedded Systems & Applications*. 83–90.

GAI, P., ABENI, L., GIORGI, M., AND BUTTAZZO, G. 2001. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*.

GAMSA, B., KRIEGER, O., APPAVOO, J., AND STUMM, M. 1999. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*. USENIX Association, Berkeley, CA, 87–100.

GOYAL, P., GUO, X., AND VIN, H. M. 1996. A hierarchial CPU scheduler for multimedia operating systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*. ACM, New York, 107–121.

HOHMUTH, M. AND HÄRTIG, H. 2001. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the USENIX Annual Technical Conference*. 217–230.

LIEDTKE, J. 1995. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM.

MEHNERT, F., HOHMUTH, M., AND HÄRTIG, H. 2002. Cost and benefit of separate address spaces in real-time operating systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*.

MOGUL, J., RASHID, R., AND ACCETTA, M. 1987. The packer filter: an efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP'87)*. ACM, New York, 39–51.

MOGUL, J. C. AND RAMAKRISHNAN, K. K. 1997. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst. 15,* 3, 217–252.

PAGH, R. AND RODLER, F. F. 2001. Cuckoo hashing. In *Proceedings of the 9th Annual European Symposium on Algorithms*. Lecture Notes in Computer Science, vol. 2161, 121–144.

PARMER, G. 2010. The case for thread migration: Predictable ipc in a customizable and reliable OS. In *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'10)*.

PARMER, G. AND SONG, J. 2010. Customizable and predictable synchronization in a component-based OS. In *Proceedings of the International Conference on Embedded Systems and Applications*.

Parmer, G. and West, R. 2007a. 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 (RTAS'07)*.

Parmer, G. and West, R. 2007b. Mutable protection domains: Towards a component-based system for dependable and predictable computing. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS'07)*. IEEE Computer Society, Los Alamitos, CA, USA, 365–378.

Parmer, G. and West, R. 2008. Predictable interrupt management and scheduling in the Composite component-based system. In *Proceedings of the 29th IEEE International Real-Time Systems Symposium (RTSS'08)*. IEEE.

Parmer, G. A. 2009. Composite: A component-based operating system for predictable and dependable computing. Ph.D. thesis, Boston University, Boston, MA.

Patil, A. and Audsley, N. 2005. Implementing application specific rtos policies using reflection. In *Proceedings of the Real Time and Embedded Technology and Applications Symposium*. 438–447.

Rajkumar, R., Juvva, K.,Molano, A., and Oikawa, S. 2001. *Readings in Multimedia Computing and Networking*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 476–490.

Regehr, J. and Stankovic, J. A. 2001. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*. 3–14.

RTLinux. Real-Time Linux: http://www.rtlinux.org.

Ruocco, S. 2006. User-level fine-grained adaptive real-time scheduling via temporal reflection. In *Proceedings of 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. 246–256.

Ruocco, S. 2008. A real-time programmer's tour of general-purpose l4 microkernels. *EURASIP J. Embedd. Syst*.

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

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

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

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

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

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

Von Behren, R., Condit, J., Zhou, F., Necula, G., and Brewer, E. 2003. Capriccio: Scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*.

Zhang, Y. and West, R. 2006. Process-aware interrupt scheduling and accounting. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE, 191–201.

Received February 2010; revised August 2010, January 2011; accepted May 2011