HIRES: a System for Predictable Hierarchical Resource Management

Gabriel Parmer
Computer Science Department
The George Washington University
Washington, DC
gparmer@gwu.edu

Richard West
Computer Science Department
Boston University
Boston, MA
richwest@bu.edu

***Abstract*—This paper presents HIRES, a system structured around predictable, hierarchical resource management (HRM). Applications and different subsystems use customized resource managers that control the allocation and usage of memory, CPU, and I/O. This increased resource management flexibility enables subsystems with different timing constraints to specialize resource management around meeting these requirements. In HIRES, subsystems *delegate* the management of resources to other subsystems, thus creating the resource management hierarchy. In delegating the control of resources, the subsystem focuses on providing isolation between competing subsystems.**

To make HRM both predictable and efficient, HIRES ensures that regardless of a subsystem's depth in the hierarchy, the overheads of resource usage and control remain constant. In doing so, HIRES encourages HRM as a fundamental system design technique. Results show that HIRES has competitive performance with existing systems, and that HRM naturally provides both strong isolation guarantees, and flexible and efficient subsystem control over resources.

I. INTRODUCTION

The general trend for processors is toward the increased computational capability of individual chips. As single chips decrease in cost and increase in processing ability, there is motivation to consolidate many applications and subsystems onto a single chip. However, the temporal constraints and resource management requirements of these applications might differ significantly, making it difficult to integrate them onto the same system. Additionally, the applications and subsystems that were physically isolated previously, must now be isolated by the policies and mechanisms of the system as they contend for the system's resources. The system wishes to satisfy contradictory goals: to provide strong isolation guarantees to different subsystems and applications, while also enabling specialized management of resources to meet the demands of each individual application.

Complicating isolation, there is an incentive for applications to aggressively specialize resource management around their requirements. For example, specialized resource management policies enable increased predictability [1], [2], [3], [4], [5], and increased efficiency [6], [7]. System consolidation raises the question of how specialized policies can concurrently be supported on a single system. In attempting to support many applications on a single system, previous work has focused on providing *temporal isolation* between them. That is, given a resource supply from the system, the resource consumption and management characteristics (e.g schedulability) of each application should be parameterized independent of its siblings. To enable application-specific resource management policies in a general system, we also require *spatial isolation*, or the inability of different subsystems to inadvertently of maliciously disrupt the execution and data of another subsystem. A subsystem's resource management policies must not be able to disrupt (temporally or spatially) another subsystem.

In this paper, we investigate the fundamental system architecture that enables the abstraction of resource management policies across different subsystems. We present our HIRES system that is fundamentally structured around HIerarchical RESource management. HIRES enables different subsystems to define application-specific, *untrusted resource managers* that control the CPU, memory, and I/O processing. We focus on these three low-level resources as higher-level resources (such as file-systems and networking stacks) are implemented on top of them. HIRES uses a hierarchical model in which *parent* subsystems *delegate* their resources to their *children*. *Subsystems* consist of a set of resource managers, and some functionality or collection of abstractions. Subsystems can be as general as the best-effort and hard real-time subsystems of an open real-time system, or could be as fine-grained an individual applications or different parts of an application. When a parent delegates resource management to a child, it empowers the child to manage that resource according to its own policies. The *root* subsystem has access to all system resources and delegates them out to its children.

The focus in HIRES is on building a system that both *predictable* and *practical*. To be predictable, the protocol for resource management delegation itself must be predictable. To be practical, the usage of resources within a subsystem must be as efficient as it is within the root. This is required to encourage the hierarchical composition of systems, and effectively encourages resource management *abstraction*. Just as functions abstract the details of a computation, resource delegation abstracts the management of resources. This abstraction is powerful as it encourages a separation of concerns whereby parent subsystems focus on providing mutual guarantees between children based in isolation, while children focus on best utilizing and allocating those resources. Additionally, it enables flexibility of design whereby child subsystems are free to manage their delegated resources according to specialized policies. This is especially useful in real-time and embedded systems where temporal constraints (thus policies) determine correctness, and the management of I/O and memory must often be cognizant of hardware limitations. A fundamental tenant of HIRES is that this abstraction must not impose any undue overhead. For the system to be both predictable and practical, the overheads of managing and using resources at level N must be equivalent to $N - 1$.

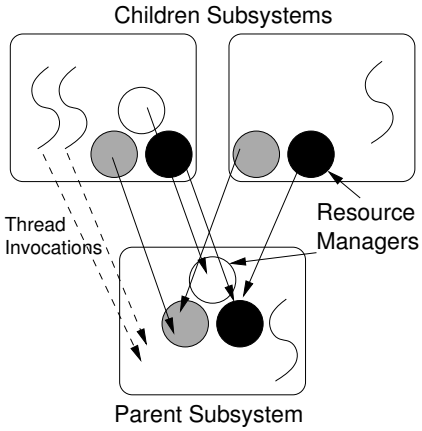

Fig. 1. A hierarchy of subsystems with resource managers. Child resource managers are delegated resources to manage from parent resource managers. Subsystems interact via thread invocations, and are memory-isolated using hardware (i.e. page-tables).

HIRES does not focus on the issue of the composability of different resource management policies. Instead we focus on the *mechanisms* able to create a predictable and efficient HRM. This work is complementary to theoretical work, for example, on scheduling composability [8], [9], [10].

The main *contributions* of this paper follow: (1) This paper investigates and introduces the HRM model as a fundamental system structuring tool based on the abstraction and delegation of resource management using untrusted, customizable resource managers. (2) We identify a small set of system goals sufficient to implement a predictable and practical HRM system, HIRES, and we detail its implementation. (3) We experimentally investigate HIRES in a variety of situations and show both the capabilities and, in some cases, the overheads of HRM.

This paper is organized as follows: Section II discusses the goals for a predictable HRM system while Section III details the implementation of HIRES. Section IV experimentally evaluates this system with respect to the goals. Section V discusses related work while Section VI concludes.

II. HIRES GOALS

Here we outline the goals of a system for general hierarchical resource management that enables application-specific customizability and predictable parent control.

G1: Efficient and predictable subsystem resource access and control. Accessing and controlling resources within subsystems must be both efficient and predictable. This goal is required to make HRM practical as a system structuring technique. A subsystem tested independently should execute predictably if placed at a different location in the hierarchy (modulo different resource allocations from parent subsystems). The *use* or *access* of a resource should not carry overhead that is a function of the depth of the subsystem in the hierarchy. For example, the cost of memory access, or CPU execution should not be dependent on the position in the management hierarchy as is the case for virtual machines that require shadow paging or binary translation [11]. Additionally, the control of resource allocation (i.e. mapping memory, scheduling threads, and receiving hardware interrupts) must also be efficient and predictable independent of the subsystem's depth, a situation we call *short-circuiting the hierarchy*. This enables system overheads such as context switch time to be considered in system analysis (e.g. schedulability analysis) independent of how that subsystem is managed and isolated in a specific HRM system.

This goal is based on what we consider a fundamental tenant of any HRM system: abstraction of resource management should not cause undue overhead.

G2: Accurate and fine-grained control over the delegation of resource management. Subsystems must be able to delegate the management of resources to their children, and they must be able to accurately track the resulting resource usage of each child. Concretely, a parent must be able to predictably and accurately delegate scheduling of a child subsystem's threads to it, enable the child to map and allocate memory amongst its own subsystems, and to delegate to the child the timely processing of interrupts that signal data transfer destined for that child. The parent must maintain accurate (page/cycle granularity) accounting information on child resource consumption. Controlled delegation and accurate accounting together enable parent subsystem's ability to make hard resource guarantees and limitations on its children subsystems. Traditional real-time techniques such as *admission control*, *resource reservation*, and *resource revocation* are naturally supported across subsystem layers. One intention of enabling parent subsystems to strictly partition their resources between children is for each sibling to be *temporally isolated* from each other, thus *independently analyzable*. Given a resource supply from the parent, the resource consumption and management characteristics (e.g. the schedulability) of each child should be parameterized independent of its siblings. We rely on previous theoretical work [8], [9], [10] for the analysis, and this paper primarily investigates how to provide system mechanisms to enable temporal isolation in a HRM system.

G3: Configurable resource managers and subsystem isolation. A fundamental goal of HIRES is that subsystems must have the freedom to define custom policies that manage resources delegated to them. We say the system supports *configurable resource managers*. Each subsystem, then, uses policies that optimally manage the resources for that subsystem's goals. We assume for generality, and to encourage both fault tolerance and security, that subsystems are possibly malicious or buggy. Though child subsystems must trust their parent to correctly delegate resource management, a child's faulty management of resources or erroneous behavior must not be able to negatively effect its parent, or siblings. Thus child subsystems are untrusted, while also giving them the freedom to most effectively manage their delegated resources. This implicitly requires the system to provide *spatial isolation* of subsystems from each other: the memory accessible from a child subsystem is by default disjoint from that of other subsystems. A practical goal in supporting configurable

resources managers is to also enable the reusability of specific policies. HIRES supports this as resource managers in different subsystems are binary compatible; the same object is reused in subsystems as required.

G4: Mediated interactions and resource sharing between subsystems. We make the assumption that parent subsystems want complete control over the delegation of resources to its children. This implies that a child should not have multiple parents, as this would prevent each parent from having a global view of the resources delegated to the child (a resource management version of the confused deputy problem). Thus, in HIRES, we construct the resource management hierarchy is a tree rather than a general graph. However, child subsystems must be able to share resources where appropriate (e.g. for shared memory), and where this is appropriate, the parent subsystem provides that functionality e.g. by implementing a memory management component that enables shared memory.

When the parent must be involved to share a resource between siblings, a degree of *HRM overhead* is inevitable. Instead of being able to manage the resource being shared directly, a child must request that the parent appropriately share the resource. This has an overhead of at least the cost of making a request to the parent. An example of HRM overhead concerns the timer interrupt: the root subsystem must have access to timer interrupts to multiplex the CPU between children subsystems, but all child subsystems also require access to notifications of the passage of time. Thus, although timer interrupts are delivered to the root subsystem, they must be propagated to the descendents. This propagation, though predictable, is the HRM overhead for delivery of timer notifications. If each subsystem could be connected to a different hardware timer, this sharing would be removed.

HIRES encourages sharing only when absolutely required. However, some resource sharing is inevitable. As children subsystems make requests (for resources or other services) from their parent, the child threads making the requests will access parent data-structures concurrently that are shared between requests. Though these structures are spatially isolated from the children, this access to a shared resource threatens unbounded priority inversion that must be prevented. The solution requires resource sharing protocols in the parent, but is additionally complicated by the child scheduling of threads.

III. HIERARCHICAL RESOURCE MANAGEMENT IN HIRES

A. Resource Allocation Coordination

Though this paper focuses on the mechanisms that enable the predictable and efficient delegation of resource management across subsystems that satisfies **G1-G4**, in this section we focus on the generic interface enabling parent and child subsystems to reserve resources, thus to assign resource management and access privileges. Parent subsystems can provide any interface for children to request resources, but we have found that having a generic interface provides a common means of communicating requirements and allocations between subsystems. Each parent exports this interface to its child subsystems.

When resources are required, a subsystem makes reservations for itself or its children from its parent. When resources are requested, admission control can be conducted in the parent to determine if the request can be satisfied. A successful reservation will enable the delegation of the management of a fixed amount of resources to the child, but does not conduct the actual delegation. Instead, it informs the resource managers (for CPU, Memory, and I/O) how much more resources can be requested by the child. This separation of concerns enables the resource reservation components to focus on splitting resources between children, while the resource managers control the policy of how those resources are allocated and delegated. Reservations are made so that the possibly costly process of admission control, when required, is conducted only when the reservation is created. The capability to associate a set of resources with a subsystem out-of-band with the usage and management of that resource is essential as it enables that assigned resource to be freely manipulated by the subsystem without interactions with the parent (**G1**).

HIRES, supports two types of reservations: *hard* and *soft reservations*. Hard reservations ensure a strict partitioning of resources whereby access to and control over a specific subset of the parent's resources are reserved to be delegated to the child. The amount of resources is negotiated and the child subsystem is guaranteed exactly that allocation. Soft reservations, on the other hand, represent resources that are delegated to a child subsystem, but that can be *revoked* by the parent (via coordination with the child). Soft reservations, then, are most useful for best-effort subsystems or to augment hard reservations to better utilize resources.

Operation	Description
`res_t create(rtype_t, rfam_t)`	create handle to an empty reservation
`void delete(res_t)`	destroy reservation handle and revoke any resources bound to it
`int bind(res_t, rspec_t)`	bind specified resources to reservation
`rspec_t wait(res_t)`	wait for an update on a soft reservation

TABLE I
THE MAIN FUNCTIONS IN THE RESOURCE RESERVATION AND NEGOTIATION API. `rtype_t` IS EITHER `SOFTRES`, OR `HARDRES`, `rfam_t` IS THE RESOURCE FAMILY (MEMORY, CPU, OR I/O), AND `rspec_t` IS A GENERIC SPECIFICATION OF A RESOURCE ALLOCATION.

HIRES reservations are similar to reservations in previous work [10], and the abstraction for creating and updating reservations is depicted in Table I. However, each subsystem implements its own independent reservation manager, augmenting the HRM with hierarchical reservation support. This imposes some constraints on how the API can be used. First, the amount of resources in a hard reservation that a child subsystem, i, makes to its children cannot exceed the hard reservations the parent makes to i, minus HRM overheads. The HRM overhead in this case is the memory consumption to maintain i's data-structures (e.g. the runqueue), or the

timer notification propagation overhead. Second, changing the resources allocated to a reservation is an operation that can involve many levels in the hierarchy. When a child delegates soft reservations to its children, and it receives a revocation notice from its parent, and if it can't relinquish resources, it must propagate the revocation notice to its children.

B. HiRes *Implementation in* Composite

We present a prototype implementation of HiRes that satisfies **G1-G4** in our Composite component-based OS [12]. In Composite, OS policies typically found in the kernel are instead defined in user-level components that can each be in separate protection domains. Each component is an implementation of a policy or abstraction whose functionality is accessible by other components through a well-defined functional interface. As threads make invocations between components, the same schedulable entity continues execution from the caller into the callee (that is, invocations use a form of IPC called *thread migration* [13]). Mutable Protection Domains (MPD) [12] enable protection domain boundaries between components to be *dynamically* added or removed to trade-off fault isolation for performance (invocations between protection domains are mediated by the kernel and incur overhead).

The Composite kernel defines a small set of low-level abstractions: threads, components (which can be schedulers that have the ability to dispatch between threads), physical memory and mapping, MPD, I/O sources, and capabilities that allow invocations between specific components. All higher-level mechanisms, abstractions, and policies are implemented in components including scheduling [14], event management, synchronization [15], and networking.

Subsystems, as discussed previously, are defined as a collection of one or more components. Each subsystem can include its own resource managers (RMs) for scheduling its threads, mapping and managing its memory, and for processing device events. Additionally, each subsystem includes a component for managing resource reservations that implements the API in Table I. In this paper, we use MPD to provide spatial isolation at the subsystem granularity, as required by **G3** (i.e. all components in a given subsystem are in the same protection domain, but interaction between subsystems requires cross protection-domain invocations). Invocations are only possible between specific components if a *capability* [12] exists denoting permission to make the invocation. Capabilities are used to constrain communication between components, and to force subsystem interactions and resource delegation to take the form of a tree (thus partially satisfying **G2** and **G4**).

Component-Based Scheduling. Composite enables the user-level component-based definition of scheduling policies. Specialized policies for QoS-aware scheduling and interrupt execution have been shown to avert livelock under heavy interrupt load [14]. Even in throughput oriented applications such as webservers, the component-based scheduling mechanisms of Composite are efficient enough for the system to be competitive with industry counterparts [12]. This paper relies on the preexisting mechanisms for component-based scheduling in Composite, but extends them as appropriate to provide predictable hierarchical scheduling in HiRes. In this section we review the mechanisms for component-based scheduling in Composite. For more details, see [14].

Thread Dispatch. The Composite kernel does not provide scheduling. Instead, user-level *scheduling* components are allowed to dispatch between threads[1]. Towards this end, the `switch_thread(thdid,flags)` system call provides this ability to multiplex the CPU. Semantically, the current thread's state is saved, and the thread, τ, identified by `thdid` is loaded and executed. If τ was previously preempted by an interrupt while executing in component C, switching to it will result in an automatic switch to C's protection domain.

Schedulers export an interface that includes functions to block and wake up individual threads. This basic support is used to implement wait-queues, event notification, and higher-level synchronization primitives such as locks with priority inheritance and priority ceiling [15]. As all critical sections in a subsystem are arbitrated using priority inheritance, sharing between sibling requests *within* a parent subsystem avoids unbounded priority inversion (required for independent analysability in **G2**, and controlled sharing in **G4**). The details of how priority inversion is provided in a scheduling policy agnostic manner is discussed in Section III-C.

C. HiRes *Hierarchical CPU Management*

A hierarchy of component schedulers is created explicitly. The root scheduler is named at boot-time, and it grants scheduling abilities to other components by making them child schedulers. The kernel ensures that the scheduling hierarchy is a tree.

To short-circuit the hierarchy, and provide as efficient scheduling control in children as in parents, each scheduler is permitted to dispatch (via `switch_thread`) between threads that have been assigned to it. Thus, as multiplexing the CPU is as efficient in a child as for a parent, this satisfies **G1**. For clarity, we contrast this strategy with a system in which child schedulers ask their parent to switch to specific threads. To switch between threads, such a system would require a number of subsystem invocations commensurate with the depth of the hierarchy.

Sibling subsystems 1) cannot make invocations to each other (as they don't have the capabilities), 2) are spatially segregated by hardware protection mechanisms, and 3) can only dispatch their own threads. This combination ensures that untrusted child schedulers cannot interfere, maliciously or accidentally, with their parents, or their siblings, yet can still define customized scheduling policies. This helps satisfy **G3** for the CPU resource. This is in contrast to more traditional hierarchical scheduling frameworks whereby scheduling policies are all implemented in the kernel where they have access to all system resources and can trivially interfere with each other.

[1]Each scheduler can only dispatch to or away from threads that have been *granted* to them by the parent scheduler. The root scheduler can dispatch all threads.

Predictable Child/Parent Coordination.

HIRES requires that parent schedulers accurately and predictably delegate resource management to children schedulers (**G2**). This section describes the protocol we have implemented to satisfy this requirement. The proposed protocol is additionally used to coordinate between parent and child by notifying the child of parent events such as the passage of time, or a child's thread blocking or waking up in the parent (e.g. waiting for I/O provided by the parent). All mechanisms and protocols discussed in this section are implemented in a scheduling library to take the burden off of the scheduling policy implementer.

Child subsystem representation. A fundamental design decision in implementing a parent scheduler is how to represent a child subsystem. The root scheduler that supports Application-Level (library-based) Scheduling (ALS), or Virtual Machines (VMs) represents all computation in the child subsystem as a single thread. All threads in the ALS or VM are multiplexed on top of a single parent thread. This has the unfortunately consequence that if the parent thread blocks, all child threads are blocked – behavior incompatible with **G1**. In HIRES, we allow parent subsystems to provide complicated higher-level abstractions to child subsystems that might require blocking, such as advanced services for networking and file-system access. Thus, since the parent subsystem can block child threads, they should not be represented in the parent as a single child thread. In contrast, if each child thread is represented in the parent's runqueue by a separate thread, it is not clear how the parent should know which thread to execute to activate the child scheduler. Hybrid models [16] use a dynamic pool of parent threads to execute the child threads. It is not clear, in the case, if such a model could be implemented predictably (the parent thread pool requires dynamic memory allocation), or how it would generalize to deep hierarchies.

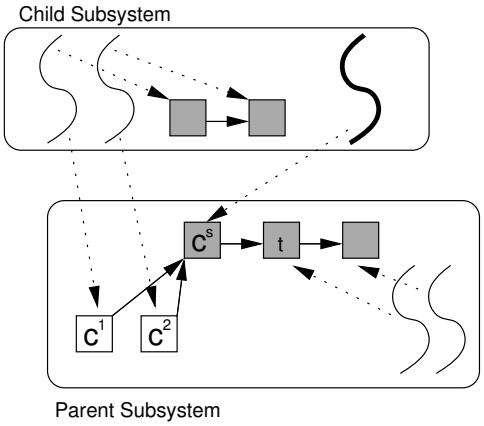

Fig. 2. Two subsystems, threads, and scheduler data-structures. The grey boxes are the data-structures for tracking threads and form the run-queue used to schedule. The dotted lines indicate the association between a thread and the scheduling data-structure. The dark thread is the child scheduler thread. No other child threads are in the runqueue (white boxes). Instead, if one of them blocks or wakes in the parent, they contain a pointer to the child scheduler thread and it is used to deliver a notification to the child. The child scheduler only schedules its threads.

HIRES, uses a combination of the previous approaches by including a single thread per child subsystem that is used to convey events to the child *and* a thread structure per child thread. The parent maintains a *child scheduler thread* (per CPU), c^s, for each child, and it is used to activate the child subsystem. When the parent wishes to delegate scheduling to the child, it will switch to c^s which is used to deliver events regarding the passage of time, or child thread block and wakeup events. c^s makes invocations from the child to the parent to retrieve these events, and when all events are delivered, the child makes a scheduling decision and switches to the appropriate thread.

In addition to c^s, a parent scheduler maintains a thread structure to track each individual child thread (c^1 and c^2). These threads are never placed into the runqueue and the parent does not directly schedule them. Instead, they include a link to c^s. This is depicted in Figure 2. When one of these threads blocks or wakes up, the parent scheduler follows that link, and places c^s into the runqueue. When executed it will deliver the child thread's event to the child scheduler. Each of these child threads can block independently in the parent without adversely effecting each other.

Resource sharing protocol. When child thread invoke components in the parent, their threads access shared data-structures (e.g. file cache, timer queues). Though these accesses are under the control of the parent (i.e. they are executing parent code), child service requests must avoid unbounded priority inversion due to these shared resource accesses. This is necessary to provide temporal isolation between siblings, and of the parent from its children. However, the child controls the timing properties of its threads by scheduling them directly (**G1**). This creates a difficult situation where the parent is accessing mutually exclusive resources in a child thread that is scheduled by the child. If the child subsystem switched away from a thread while it happened to be executing in the parent, and holding a resource, unbounded priority inversion is possible. Thus we have a conflict between subsystem resource control **G1** and subsystem temporal isolation **G2**. In HIRES, we make a compromise: child subsystem control their threads at all times (though they can be preempted if a parent chooses), unless they are holding a resource while executing *in* the parent subsystem. In this case, the parent scheduler uses priority inheritance in such cases to ensure predictability.

Traditional implementations of priority inheritance are awkward in HIRES. As child thread structures are never placed in the parent's runqueue (i.e. the parent doesn't schedule individual child threads), increasing the priority of these threads is insufficient. Instead, to implement priority inheritance, each thread structure in the parent scheduler maintains a *dependency* link that is set to the (possibly child) thread that holds a requested resource. When the dependent thread contests the mutually exclusive resource, it remains in the runqueue, and if the scheduling policy chooses for it to execute, the scheduler will track the dependency links and switch to the depended on thread first, to ensure it finishes its critical section in a bounded manner. The use of dependency links is illustrated in

Figure 3. Locks with priority inheritance within the parent are implemented in a synchronization component [15], by using an interface provided by the scheduler that allows the current thread's dependency links to be set to the holder of a contended lock.

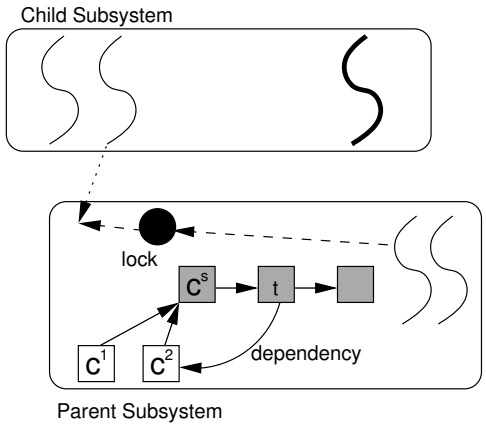

Fig. 3. Thread c^2 from the child subsystem invokes the parent, and holds a shared resource (the black circle is a lock). One of the parent's threads, t, contends the lock. Though c^2 is not in the runqueue of the parent scheduler, t's dependency pointer will result in the parent running c^2 as in priority inheritance.

HIRES enables children subsystems to control the scheduling of their threads. However, in rare circumstances the parent does maintain scheduling control over child threads: when they are holding shared resources and executing in the parent. Importantly, this does not adversely impact the schedulability of child subsystems. The maximum resource hold time while executing within the parent, or the maximum interference between siblings, is bounded and a parameter of the maximum critical section hold time *in the parent*, not of the each sibling.

Timekeeping in child schedulers. Keeping accurate real-time in child schedulers requires the parent scheduler to notify its children of the passage of time. Specifically, each scheduler in the system maintains its own list of timing events corresponding to when its threads wish to be woken up. This enables each subsystem to define its own time-related functionality. Each child scheduler publishes to its parent the *next* wakeup time instead of all future wakeups. Timing events are initiated by a clock tick and propagate up the hierarchy. The parent delivers a timing event to a child that includes the number of ticks that have occurred since the child was last executed. This enables each child scheduler to maintain an accurate representation of the real passage of time (required for **G1**).

Protocol for child event delivery. In traditional hierarchical scheduling systems, function invocations are used to pass events amongst schedulers as all schedulers are trusted and in the same protection domain (e.g. in the kernel). **G3** prevents parents from invoking children as such invocations might never return, or might fault. This is another reason for coordinating between parent and child using child scheduler threads. All execution time spent in the child scheduler thread is accounted to the child subsystem. In fact, in HIRES, all execution time for all child subsystem threads is accounted to c^s to make time-keeping simple in the parent.

The details of the protocol for event delivery and coordination between parent and child are described here. A child scheduler repeatedly asks the parent for events, and also specifies if there are runnable child threads (i.e. if the parent doesn't deliver an event, should child subsystem be blocked?). Each time the parent is invoked by the child, it returns an event, and tells the child if there are further events. When the child processes all its events, it switches to the next thread according to its policy. Events (timer events and I/O) arrive asynchronous to the delivery of events to child schedulers. This causes a race condition that can prevent events from being delivered to the child in a predictable manner: 1) The parent returns the last event to the child and the child is about to dispatch its running thread, c^1, 2) an event is delivered to the parent, and the parent returns to c^s to deliver it, 3) the child continues execution where it was preempted and switches to c^1, instead of retrieving the event from the parent. Now c^1 is executing when the event might signify the waking of c^2, a higher priority thread.

To prevent this situation in HIRES, we modify the COMPOSITE kernel so that when a child calls `switch_thread` to dispatch a thread (c^1 in this example), the kernel checks to see if a parent wishes to deliver an event (i.e. if a `pevt` variable in child is set). If so, it returns an appropriate error code designating there are still pending events. When a parent dispatches a child scheduler thread, it passes a flag that sets `pevt` for that child scheduler. This is the only synchronization needed to ensure a protocol for the timely delivery of parent events (**G2**), and does not require the parent trust the child.

D. HIRES *Hierarchical I/O Processing*

In HIRES, the I/O resources being managed are (1) the bandwidth of the specific I/O device, e.g. the transmission bandwidth for a networking card, and (2) the delivery of processor interrupt notification for a device to the subsystem that should process the event. In COMPOSITE, we currently implement device drivers in the kernel (we use Linux device drivers via our Hijack [1] technique). Thus interrupts trigger in the kernel, and a minimal amount of processing (mainly in the device driver) is conducted. If the interrupt was delivered due to data delivery (e.g. via DMA), HIRES must determine which subsystem to notify, and pass it the data. As HIRES wishes to short-circuit the hierarchy (**G1**), this interrupt-time notification should be delivered directly to the subsystem that will process the data and involves the *activation* of an event thread in that subsystem. Thus, HIRES must consider the distribution of device interrupt notification as a hierarchical resource management decision. The main difficulties are 1) how to schedule the execution resulting from the interrupt which requires scheduling decisions from multiple schedulers in the hierarchy without the costly operations of activating those schedulers (again, to ensure we meet **G1**), 2) how to identify from the data accompanying the interrupt which event thread in which

subsystem should be activated, and 3) how to maintain proper accounting information for all parent schedulers to ensure that the event thread's execution is charged to the proper child subsystem.

Interrupt Execution Scheduling: Interrupts can occur at a high frequency (*e.g.* receiving packets from a GigE network), and HIRES must minimize the overhead of scheduling the event processessing resulting from each interrupt. Additionally, because interrupts destined for processing in a child subsystem would intuitively require parent scheduling decisions, HIRES must find a way to short-circuit the hierarchy to avoid directly involving multiple parent schedulers.

To avoid invoking the schedulers for every interrupt, yet still delegating all policy decisions to the scheduler, COMPOSITE uses the shared scheduler-kernel memory region to share information regarding thread and event thread priority, state, and execution time. The scheduler publishes to this region current thread priorities, and the priority of any event thread should it become active. When a subsystem's event-thread activates, the kernel compares its priority in this region to that of the currently executing thread. It does this for all schedulers that the preempted thread and the event thread share. If they all determine that the event thread is of higher priority, it is dispatched automatically without directly invoking the schedulers. Note that this *does* incur overheads proportional to how many scheduler's shared structures must be investigated. These costs include two cache misses and one TLB miss per scheduler in the worst case. We consider this an acceptable solution. When an event thread completes execution, the kernel will automatically switch back to the previously executing thread unless another has been awoken or any priorities have changed in the mean time.

The shared region is used by the kernel to publish execution times and event thread execution states to each scheduler. Parent scheduler components maintain control over temporal policy (**G2**) by dictating the importance of all threads at all points in time, while avoiding the overhead of costly scheduler invocations for each interrupt (thus satisfying **G1**).

I/O HRM Case Study: Networking. To demonstrate how I/O is hierarchically managed, we have implemented networking support in HIRES. When a NIC DMAs data into main memory and triggers and interrupt on the CPU, the device driver executes in interrupt context and when the transferred data is accessible, a lightweight classification engine processes areas of interest in the packet header. This is used to identify which subsystem should process those packets (i.e. the subsystem associated with packets with a specific destination port). The data is copied into a ring-buffer shared in the appropriate subsystem, and its interrupt thread is activated. As scheduler invocations are avoided in the common case, processing of received packets is as efficient for deeply nested resource managers as for the root. This *early demultiplexing* of data received from the I/O device enables the efficient processing of interrupt-triggered I/O and separates a shared interrupt source into many classified events (**G4**). We build the classification engine into the kernel of HIRES.

In addition to the reception of I/O data, HIRES must control the sending of data. The current prototype requires that a single component that interfaces with the kernel-level device driver must transmit all data. In this component, we implement the following policies: (1) when a thread from a subsystem attempts to transmit a packet, it is confirmed that the source port in the packet has been bound (using `bind`) to that subsystem, and (2) it can rate-limit the amount of data sent from a specific subsystem (in accordance to how much bandwidth has been bound to that subsystem).

Resource control and accurate accounting of resource allocation (**G2**) is maintained as only parents can bind ports and bandwidth allocations to child subsystems. HIRES provides an interface for parent subsystems to retrieve bandwidth usage information per child subsystem.

HIRES I/O processing generalization. The demultiplexing layer in the kernel must be implementable for different forms of I/O. Though we believe that many I/O sources can be demultiplexed in a timely manner, not all interrupt sources can. For example, the timer interrupt must be delivered to the root subsystem to maintain isolation and system-wide time-keeping. However, all subsystems require some notion of time, so the timer information is propagated up the hierarchy. We detail the costs of this in Section IV.

E. HIRES *Hierarchical Memory Management*

Memory is the most straightforward of the resources to manage hierarchically. Virtual address space mappings (e.g. provided by page-tables) ensure that memory accesses are as efficient at any level in the hierarchy (**G1**). By enabling the parent to control the mappings for any subsystem, each parent has accurate control over resource allocations (**G2**). By requiring all memory requests (`binds`) to make cross protection-domain invocations, each child is spatially isolated from its parent, thus for its siblings as well (**G3**).

Operation	Description
`int alias(paddr_t, caddr_t, cid_t)`	create a mapping that aliases a parent address to a child address
`void remove(caddr_t, cid_t)`	parent removes aliases from child's subtree

TABLE II
HIRES MEMORY MANAGEMENT API. A PARENT INVOKES THESE FUNCTIONS TO GRANT MEMORY TO THE CHILD (IDENTIFIED BY ITS ID `cid_t`), AND REMOVE IT.

As efficient memory access is enabled by the hardware mechanisms, here we focus on the HIRES mechanisms for resource delegation and assignment which are influenced by the memory management in L4 [17]. The COMPOSITE kernel provides a simple system call that is used to map a given physical frame into a component at a specified address. A single component is permitted to use this low-level API, and it exposes a simple interface, introduced in Table II, through which memory is mapped appropriately from parents into children. `alias` and `remove` are similar to `map` and `unmap`

in L4 [17]. `alias` maps a page in the memory management component into another at a specific location, while `remove` will unmap *all* aliases rooted at the parent. We diverge from the L4 mechanisms in two main ways: (1) by not providing a `grant` call that not only adds a mapping to the child, but *also* removes it from the parent (this breaks removes parent control over that memory page and breaks **G2**), and (2) by integrating the implementation of `alias` and `remove` with the resource reservation API so that memory can only be mapped into a child if amount of memory has been bound (via `bind`) to the child. Additionally, in contrast to L4, we implement the logic and data-structures to track memory mappings in a user-level component as opposed to in the kernel.

Sharing of memory between siblings is mediated by the parent. The parent provides an interface for aliasing physical frames between multiple siblings. However, such sharing lessens the isolation between siblings, and we have found that, in most cases, sharing is only required between child and parent. We use locks that implement priority inheritance to avoid unbounded delays due to the sharing of "mapping" data-structures in the parent between two sibling threads [15]. This implementation of priority inheritance utilizes the explicit tracking of task dependencies as described in Section III-C. The careful control of resource sharing satisfies **G4**.

IV. Experimental Evaluation

We conduct all experiments on an Intel Atom n330 clocked at 1.6 GHz with 2 GB of memory. Only one SMT thread (and only one core) is active. We use the hijack [1] technique to boot into COMPOSITE, and we utilize the Linux 2.6.33 Atheros L1C networking device driver. This system is connected to a client via gigabit ethernet.

In this section, we wish to evaluate (1) the effectiveness of HIRES in short circuiting the hierarchy and enabling efficient execution at any depth, (2) the resource management delegation overhead in HIRES, and (3) the ability of all aspects of the system to "come together", and provide the expected resource usage performance for real application, while ensuring that the parent can provide strict isolation. First we quantify a fundamental cost in COMPOSITE, that of an invocation between protection domains (which are on the granularity of subsystems in this paper). An invocation of function f in another component (subsystem) switches page tables twice and switches between user and kernel level four times, takes 0.815 μ-seconds. Though Linux was not designed primarily for IPC, for the sake of context, a comparable operation between threads communicating over a pipe takes 10.305 μ-seconds (averaged over 10K invocations).

Efficient resource access and control. HIRES is set up so that a resource is bound to a subsystem at a specific depth in the hierarchy. We investigate the effect that an increasing depth has on the efficiency and predictability of accessing and manipulating that resource. Ideally, accessing and controlling resources from a subsystem would have a constant cost regardless of the subsystem's depth in the hierarchy, effectively short-circuiting the hierarchy and making HRMs as efficient as a single-level of resource managers.

Figure 4(a) plots the latency of sending an event from one thread to another. Two threads, A and B, utilize a component that provides event management. Thread B waits for an event, and thread A triggers it. The event component uses the scheduler (in the subsystem) to wakeup B when the event is triggered, and to put it to sleep when it waits for the next event. We measure the latency between when A triggers the event, and when B receives it. This tests how efficiently and predictably the scheduler can manage its threads. We report averages over 1500 samples (the standard deviations are always less than 0.01μs.) Figure 4(a) shows that regardless of the depth of the subsystem in the hierarchy, the scheduler is able to efficiently multiplex the CPU. The kernel automatically updates the execution time accounting information in each subsystem the threads have been granted to. We compare this to sending a single byte between pthreads in the same process in Linux.

Figure 4(b) measures the response latency between (i) when the kernel has received an interrupt from the NIC and it attempts to activate an event thread in a subsystem, and (ii) when the high-priority event thread is activated. The depth of the subsystem (thus where the interrupt is delivered) in the HRM system is varied. We are interested in any variations in response latency across depths. A client sends 450 UDP packets/second, and we take the average and standard deviation of 5000 samples. The maximum latencies are determined by interference with the higher-priority timer tick and are in the range [29.91,40.48] μs. The graph shows that there is little variation across depths, thus I/O is processed predictably, and with a low response latency regardless of the subsystems location in the HRM system.

Figure 4(c) depicts the latency for mapping, then unmapping (measured separately) a single page that was granted to the memory manager in a subsystem. We vary the depth in the HRM system. We plot the results for both an `alias` implementation that zeros the page, and one that does not. This cost is compared to the costs of mapping and unmapping a private, anonymous page using `mmap` and `munmap` in Linux. In this case, we ensure that the page is really mapped in, by writing to it. We report the average across 10K samples. As in (b), the worst case costs in HIRES are determined by interference by timer tick and range from [22.25,37.33] μs. This again shows that HRM is possible without penalizing "deep" subsystems.

HRM overheads. Here we investigate the overheads imposed by HRMs. Primarily, each subsystem unavoidably consumes memory. Thus if a subsystem requires X memory, its parent must reserve $X + O$ where O is the amount required for itself. To get a rough estimate of how much memory this could be, a fixed priority round robin scheduler, the hierarchical memory manager, network I/O interface, and resource reservation components total 102K. This gives a designer an idea of the overhead. However, different resource management policies use different data-structures and algorithms, and thus

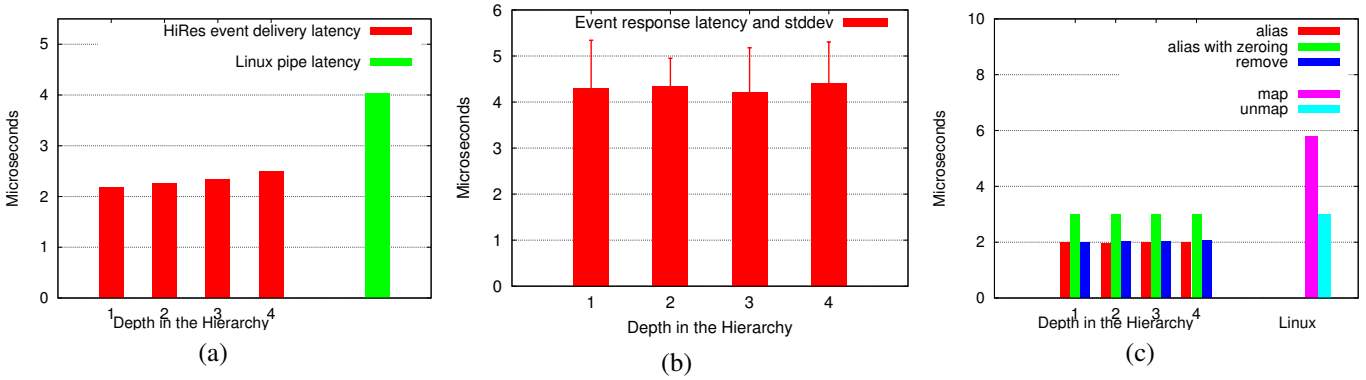

Fig. 4. The effect of hierarchical depth resource management. (a) CPU management: The latency for communication between a thread, and a second higher-priority thread in HIRES and in Linux. (b) I/O management: The latency for delivery of an I/O event to an event-thread. (c) Memory management: Overhead to `alias`, `alias` and zero, and `remove` a page.

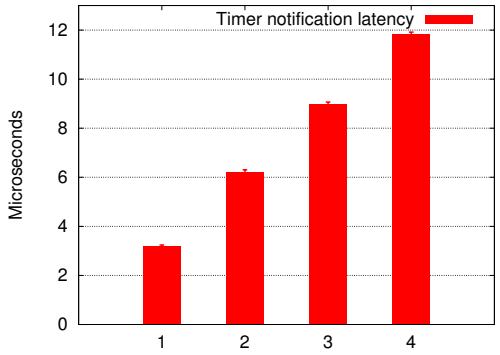

Fig. 5. Latency to receive a timer notification for different hierarchical depths.

use different amounts of memory. The overhead is specific to the implementations chosen.

An additional source of overhead comes from shared interrupt processing. Certain interrupts cannot be demultiplexed the way that the network is. For example, the timer-tick serves an important function in allowing the root subsystem to keep time, and provide the requisite isolation between children. This interrupt should be delivered to the root scheduler, but (as detailed in Section III-C), timer tick information is propagated up through the hierarchy via the child scheduler threads. This approach adds latency to time-triggered wakeups. Figure 5 investigates this latency by measuring the time between (a) when the timer interrupt occurs and COMPOSITE attempts to deliver it to the root scheduler, and (b) when a high-priority thread in a given subsystem receives a notification that the interrupt occurred as delivered by the protocol in Section III-C. Though the cost increases proportional to the depth in this case, we still consider the cost to be low enough to be practical.

Efficient, accurate, and predictable HRM. Here we evaluate the performance impact that hierarchical resource management has on a complex application (**G1**), study the accuracy of parent accounting and control over the child (**G2**), and do so while the child subsystem is spatially isolated from the parent (**G3**). We execute an application that utilizes all resources to determine if the microbenchmarks above are translated into a negligible overhead in more complex scenarios. Specifically, we use a web server that is defined by the combination of 22 components and consists of 6 separate threads, one of which is the event thread to handle packet reception. This web server has been shown [12] to have comparable or better performance to an industry counterpart[2]. The server contains components to manage its resources (a networking stack interfaces with the NIC, and the application presumes a specific fixed-priority assignment). This is the type of application that would run along-side real-time tasks in an open real-time system (e.g. webservers are often used for system configuration). Though web-servers do not have real-time requirements, HIRES requires efficient execution to be practical, and a web-server will test the ability of the system to exist in open real-time environments.

Depth	Connections/sec	Stddev	Latency	95% Latency
0	6356	205	3.5	5
1	6294	69	3.8	5

TABLE III
WEB-SERVER PERFORMANCE AT DIFFERENT LEVELS.

First, we study the efficiency of the web server when executing normally as the root subsystem, then we evaluate it executing as the child to a root subsystem. A client machine uses `ab` (see footnote 2) to maintain 20 concurrent connections over the span of 30 seconds. We average the connections per second over 30 readings. Table III compares the performance of the two setups. There is a 1% performance degradation for executing the webserver subsystem as a child. We find this overhead to be acceptable. `ab` reports the client's average connection latency (in ms) which is similar for both approaches. `ab` also reports that for both approaches, 95% of the connections took 5 ms or less. We believe this demonstrates that HIRES satisfies **G1** for a complex application.

[2]httpd.apache.org

Next we investigate the ability of a parent subsystem to accurately and predictably control its children (**G2**). The root scheduler implements a deferrable scheduler policy to constrain the execution of the best-effort subsystem (i.e. the web-server). However, using *only* servers to control CPU allocations does not help to both maintain web-server performance and to ensure predictable execution of real-time tasks. First, the web-server consists of 6 threads with many dependencies between them (e.g. the event thread passes data to the main thread, which passes data to the CGI thread). As the relative execution time of these threads is not predictable, it isn't clear how to assign reservations to their deferrable server threads. Second, the web-server assumes a specific priority assignment amongst its threads which couldn't be accommodated if e.g. the root used rate-monotonic priority assignments. Thus, hierarchical resource management here is required to both maintain the semantics of the web-server *and* to limit its interference with the real-time tasks. The web-server is run as a child subsystem accounted in the parent as a single deferrable server. Three cpu-bound threads are also scheduled in servers with reservations (budgets and replenishment periods) of 4 of 25, 3 of 20, and 1 of 10 timer ticks.

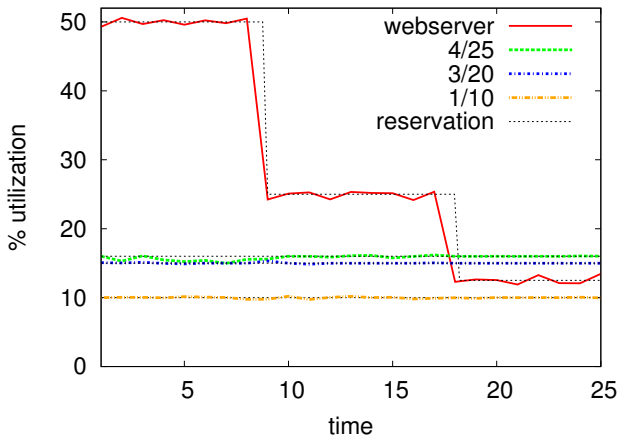

Fig. 6. CPU utilization of best-effort subsystem and real-time tasks.

Figure 6 depicts the CPU utilization of threads over time. The budget of the web-server child subsystem remains 4 throughout the test. The x-axis is broken into three phases. First, when the budget replenishment period of the web-server is set to 8, second when it is set to 16, and third when it is set to 32. For consistency, the webserver has the highest priority in all scenarios. The resource utilizations for each of the real-time threads remains consistent with their reservations, as does the web-server's subsystem. Even though the web-server is triggered directly when network packets arrive (which avoids invoking schedulers), and even though it is a highly sporadic workload, the parent is able to maintain accurate accounting information, and schedule accordingly. The web-server's throughput goes down almost exactly in proportion to the decreases in reservation, indicating that even when the best-effort subsystem aggressively scaled back, hierarchical overheads don't increase.

V. Related Work

Many systems have explored how to enable application-specific resource management policies. Exokernels [18] move functions typically found in the kernel, into user-level libraries. This approach distributes resource management decisions, enabling applications to specialize accordingly. However, without central control of system resources, the task of isolating individual applications becomes difficult. In contrast, systems such as resource kernels [10] enable the strict control over the allocated resources to each application, but do not explicitly promote the use of customizable resource managers. HiRes enables both application-specific resource management policies and parent control over allocation by providing mechanisms for hierarchical resource management.

Various other systems have attempted to support HRM of a single resource. To address CPU management, hierarchical scheduling implementations have been researched. Some form the hierarchy of schedulers in the kernel, requiring that they be trusted [19], [20]. Others use CPU donations [21], [22] to enable any thread to grant processing to any other. Though general, this makes it difficult for a parent to control the assignment of all resources to a child. Finally, two-level thread management systems [16], [23] multiplex application-level threads on top of kernel threads. Our previous work on Composite [14] provided the foundation for HiRes by enabling a hierarchy of schedulers to be formed, but it did not investigate the protocols for delegation of resource management, nor the difficulties in mediating resource sharing in the parent.

In terms of hierarchical memory management, our interface is similar to L4's [17]. However, HiRes diverges by ensuring that the mapping primitives integrate into the HRM system's reservation framework and into the delegation model (thus the omission of `grant`).

Research has been conducted that attempts to vector I/O appropriately to user-level [7] by performing early demultiplexing on device interrupts. HiRes does so while ensuring that proper accounting and scheduling is conducted throughout a hierarchy of schedulers.

VI. Conclusions and Future Work

HiRes is a system for the predictable hierarchical management of the fundamental system resources: CPU, memory, and I/O. HiRes provides abstraction over resource management by enabling policies to be implemented in the subsystem that best understands how resources should be used. This encourages the separation of concerns whereby a parent resource manager focuses on effectively providing temporal isolation between its children, and the children managers focus on best using their resources to meet that subsystem's goals. Through pervasive spatial isolation, each subsystem freely defines customized managers without endangering their parents or siblings. We outline a number of goals for the construction of a predictable

HRM system, and emphasize the importance of avoiding undue overheads in the abstraction of resource management. Results show that HIRES is effective at avoiding such overheads, both in microbenchmarks and for a complex application, while also being competitive with existing systems. Additionally, we show that parents accurately and precisely control resources delegated to their children.

The HIRES and COMPOSITE source is located at `www.seas.gwu.edu/~gparmer/composite.html`.

REFERENCES

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

[2] P. Goyal, X. Guo, and H. M. Vin, "A hierarchical CPU scheduler for multimedia operating systems," in *Proceedings of the second USENIX symposium on Operating systems design and implementation (OSDI)*, 1996.

[3] S. Ruocco, "User-level fine-grained adaptive real-time scheduling via temporal reflection," in *Proceedings of 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, 2006.

[4] "Real-Time Linux: http://www.rtlinuxfree.com."

[5] J. Liedtke, H. Haertig, and M. Hohmuth, "OS-controlled cache predictability for real-time systems," in *RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, 1997.

[6] R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer, "Capriccio: Scalable threads for internet services," in *19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[7] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A user-level network interface for parallel and distributed computing," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

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

[9] A. K. Mok, X. A. Feng, and D. Chen, "Resource partition for real-time systems," in *Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*, 2001.

[10] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein, "Analysis of hierarchical fixed-priority scheduling," in *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS)*, 2002.

[11] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2006.

[12] G. A. Parmer, "Composite: A component-based operating system for predictable and dependable computing," Ph.D. dissertation, Boston University, Boston, MA, USA, Aug 2009.

[13] G. Parmer, "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)*, 2010.

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

[15] G. Parmer and J. Song, "Customizable and predictable synchronization in a component-based os," in *Proceedings of the International Conference on Embedded Systems and Applications (ESA)*, 2010.

[16] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: effective kernel support for the user-level management of parallelism," in *Proceedings of the thirteenth ACM symposium on Operating systems principles (SOSP)*, 1991.

[17] J. Liedtke, "On micro-kernel construction," in *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, 1995.

[18] D. R. Engler, F. Kaashoek, and J. O'Toole, "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, 1995.

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

[20] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards hierarchical scheduling on top of vxworks," in *Proceedings of the Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '08)*, July 2008, pp. 63–72.

[21] B. Ford and S. Susarla, "CPU inheritance scheduling," in *Proceedings of the second USENIX symposium on Operating systems design and implementation (OSDI)*, 1996.

[22] J. Stoess, "Towards effective user-controlled scheduling for microkernel-based systems," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 4, pp. 59–68, 2007.

[23] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the art of virtualization," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.