Execution Stack Management for Hard Real-Time Computation in a Component-Based OS

Qi Wang, Jiguo Song, and Gabriel Parmer
Computer Science Department
The George Washington University
Washington, DC
{interwq,jiguos,gparmer}@gwu.edu

***Abstract*—In addition to predictability, both reliability and security constraints are increasingly important. Mixed criticality, and open real-time systems execute software of different certification and trust levels. To limit the scope of errant behavior in these systems, a common approach is to raise isolation barriers between software components. However, a thread that executes through multiple components computes on execution stacks spread across each component. As these stacks require backing memory, each component has a finite amount of execution stacks. In this paper, we treat these stacks as shared resources, and investigate the implementation of traditional resource sharing protocols in a real component-based system. We implement multi-resource versions of the Priority Inheritance Protocol (PIP) and Priority Ceiling Protocol (PCP) for these shared stacks and find – surprisingly – that neither provide better schedulability characteristics than the other for all system parameterizations. Additionally, we identify the relationship between allocating additional stacks to components, and system schedulability. Given this, we describe and evaluate algorithms to ensure system schedulability while seeking to minimize the amount of memory consumed for stacks.**

I. Introduction

Embedded and real-time systems are increasingly required to provide not only predictability, but also increased reliability, security, and isolation guarantees. Open real-time systems in which hard real-time tasks execute along-side best effort and untrusted applications require not only that the real-time tasks meet their deadlines, but that they are isolated from the possibly faulty or malicious programs. This motivates a class of systems that provide fault-isolation at a finer granularity than is typically provided by monolithic operating systems and applications. Hardware techniques for memory isolation (*e.g.* page tables) are commonly used to segregate the functionality of the system into separate *components*. A fault in one component cannot access or corrupt the memory in another, thus constraining the fault propagation and the adverse effects of buggy or malicious components. Examples of such systems include μ-kernels [1], component-based OSes [2], [3], and middle-ware systems [4].

Communication between components in such systems is typically mediated by the kernel and because memory in one component is generally inaccessible in another, the *execution stacks* of threads are distributed across all components that thread has invoked. Specifically, a thread does not use a single C execution stack (*i.e.* used to track function calls, provide scoped local memory, and an area for spilling registers) when making invocations between components; instead, a stack[1] *per-component* is used for the local computation *within* that component. Each component is allocated a fixed number of stacks, and they must be used in a mutually-exclusive manner by threads that invoke the component. Unfortunately, this presents a problem: If multiple threads concurrently invoke a component with a single stack, there is naturally *contention* on that stack – threads must block waiting for the holder (*i.e.* the thread executing in that component) to exit the component, thus make the stack available for another thread to use. This causes *blocking delays* that affect system schedulability in real-time systems. However, we observe that as more stacks are allocated to a component, the blocking delays due to contention are lessened as multiple threads can concurrently execute in the component. In the extreme case, each component in the system can be given N stacks, where N is the number of threads in the system, to remove all blocking delays. There is, then, a trade-off between the schedulability of systems with prohibitive blocking delays, and memory usage for stacks.

In this paper, we explore multiple facets of stack contention. *First*, we explore an implementation of invocations in our component-based OS, Composite, and implement common resource sharing protocols – the multi-unit Priority Inheritance Protocol (PIP), and the multi-unit Priority Ceiling Protocol (PCP) [5], [6] (henceforth, we omit the multi-unit description and simply refer to PIP and PCP for brevity). These protocols avoid the unbounded priority inversion that occurs when a high-priority thread blocks waiting for a stack from a low-priority holder while a medium priority thread executes for a possibly unbounded amount of time. We find that PCP has significant practical implementation overheads when used for stack sharing and when we apply these overheads in a system model, PIP actually has more favorable schedulability characteristics in certain circumstances. This is opposed to conventional blocking delay formulations in which PCP is superior to PIP [5]. We investigate PIP, as it is a commonly implemented protocol in existing systems [7], [8], and we investigate PCP due to its favorable analytical properties. *Second*, given the trade-off between memory usage on stacks and schedulability, we show how *blocking delays can be manipulated* by assigning additional stacks to specific components.

Contributions: The main contributions of this paper include:

[1]In this paper, we use “stacks” as shorthand to refer to “C execution stacks”.

(1) We identify the problem of stack sharing in a reliable OS architecture as a system design issue involving a trade-off between memory consumption and schedulability; (2) We formulate the problem of stack sharing as a traditional resource sharing problem, and provide analytical blocking time calculations that explicitly consider memory allocations to stacks throughout the system, and how these allocations affect schedulability; (3) We detail and study the implementation of both PCP and PIP for stack sharing in a component-based OS; (4) We show that contrary to convention, PIP can for some system configurations have better schedulability properties than PCP when the practical implementation overheads of PCP are taken into account; (5) We introduce offline algorithms to allocate stacks for system schedulability and show that a small number of stacks are necessary to ensure predictability.

Section II discusses the system implementation of component invocation in COMPOSITE to motivate the rest of the paper. Section III constructs a model of a component-based system while Section IV uses this model to define the blocking terms of tasks for PIP and PCP given specific stack allocations. Section V investigates algorithms for assigning stacks to components for schedulability and Section VI evaluates the model and algorithms in simulation. Section VII discusses related work while Section VIII concludes.

II. COMPONENT INTERACTION IN COMPOSITE

We use the COMPOSITE [2] component-based OS as a motivating system for this work. System policies and abstractions are defined as components. Components are implemented by code and data that executes at user-level. A component's functionality is accessed by other components only through a well-defined functional interface. Each component is encapsulated and opaque behind this interface, enabling them to be separated into different protection domains. These protection domains provide memory isolation by limiting the scope of accessible memory to each component, thus enabling reliability by constraining the propagation of faults. These component protection domains are provided by hardware mechanisms (page-tables).

In COMPOSITE, even low-level system services such as scheduling [9], synchronization [10], physical memory management, I/O handling, and event management are defined by replaceable components. This simple model enables complicated policies such as hierarchical resource management [11]. The OS can be customized to the requirements and goals of individual applications, and the component-granularity protection increases fault isolation and security. A functional system is the composition of a set of components. COMPOSITE includes 6K lines of code for the kernel (compared to millions of lines of code for monolithic systems such as Linux). All component code, including 3rd party libraries, totals over 100K lines of code in COMPOSITE.

Components invoke functions defined by and in the interface for other components. These invocations functionally mimic the *call* and *return* semantics of function calls. When a thread executing in a component c^0 invokes a function in the interface for component c^1, execution for that thread discontinues in c^0 and begins in c^1, until it returns and resumes execution in c^0. COMPOSITE uses a migrating thread model to implement this communication [10] as depicted in Figure 1. The same schedulable entity (thread) migrates with execution between components, and thread's execution in each component requires a separate C execution stack. Thus, in COMPOSITE there is a separation of the scheduling context (that migrates between components), and execution contexts (that provides a stack to execute on in a component). It should be noted that predictable systems that use synchronous communication between threads [7], [8] also attempt to separate the scheduling context and execution contexts while using synchronous rendezvous between separate threads.

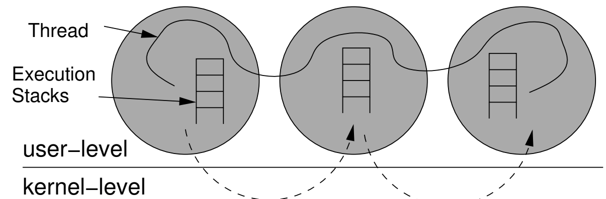

Fig. 1. Thread migration in COMPOSITE. The same schedulable entity migrates between components via invocation. Components are in separate memory protection domains, thus invocations require kernel mediation. Separate *execution contexts* (C stacks) must be used for the thread's execution in each component.

Despite the decomposition of system software into hardware-separated components, the efficiency of COMPOSITE is reasonable. In [2], we show that a web-server implemented as more than 25 components (causing over 70 component invocations per HTTP client request) is competitive with traditional software (apache on Linux). We have also studied mechanisms to dynamically remove protection domain boundaries between components [12] to trade fault isolation for performance. For the web-server, we find that removing inter-protection domain invocation overheads does improve performance, but that a system with isolated components is still competitive with traditional techniques.

A. COMPOSITE *Invocation Implementation*

When a function defined in a component's interface is invoked, the kernel mediates the communication (this is necessary as protection domains must be switched). The kernel makes an upcall into the "server" component that triggers execution at a fixed address associated with the function being invoked. This code is written in assembly as no stack has yet been found to execute on (*i.e.* C code assumes an active stack). This assembly code's first job is to locate a stack to execute on, or invoke the stack manager if there are no available stacks. All arguments for an invocation are passed in registers and to keep the kernel invocation path simple and efficient, the kernel does not copy or map memory. If a function requires more arguments than there are registers (*i.e.* there are only 6 general purpose registers on x86-32), or if the arguments are pointers to data regions, an Interface Definition Language (IDL) compiler generates stubs to pass the data in shared

memory. Shared memory regions are set up and managed by a devoted component; the details are beyond the scope of this paper.

Figure 2 depicts the implementation of invocations and execution contexts in COMPOSITE. Execution stacks are shared resources used by multiple threads when they invoke a component. Unused stacks are maintained on a singly-linked freelist. To avoid the unbounded priority inversion that can result from contention on the stacks (*i.e.* the shared resources), we summarize the implementation of both multi-unit priority inheritance and multi-unit priority ceiling protocols [5], [6].

Multi-unit priority inheritance protocol in COMPOSITE. Priority inheritance is optimistic in the sense that when a accessed resource is uncontended, the scheduler isn't involved (there are no priority changes), and management of the shared resource can be handled locally. A good example of this is mutexes that implement PIP in Linux [13]: in the uncontended case all kernel invocations (system calls) are avoided. As PIP can avoid interaction with other components, it is an appealing mechanism to integrate into an optimized inter-process (or inter-component) communication path. The common case of no contention imposes no overhead over the basic invocation costs. For this reason, previous systems [8], [7] use priority inheritance to avoid unbounded priority inversion when there is contention for execution contexts.

In COMPOSITE, when a stack is allocated to a component, it is part of a singly-linked free-list when unused. The first operation performed upon invocation is to attempt to retrieve a stack from the freelist. Instead of using an explicit semaphore to protect the freelist, we use non-blocking synchronization that can be made predictable on a uni-processor [14] (or on a multiprocessor with resource partitioning). Specifically, the assembly stub finds the freelist of stacks (which is at a known address in the component), and removes the first stack from the list. This operation is performed atomically[2] Once a stack is acquired, the thread sets its stack pointer, and the intended component function is invoked. This is illustrated by step 1 in Figure 2(a). When the component's function returns, the stack is atomically added back into the freelist, and the thread returns to the invoking component (step 2 in Figure 2(a)). Thus, in the uncontended case, the overhead for sharing stacks is minimal, including only stack freelist manipulation.

Figure 2(b) details the steps taken for PIP given contention on the stacks (*i.e.* more threads have invoked the component than there are stacks allocated to the component): 1) Low priority thread τ_l uses the stack, and is preempted by the high priority thread τ_h. 2) τ_h invokes the component, finds no stacks on the freelist, sets a "contended bit" in the component, and invokes the stack manager that suspends the thread in the scheduler, creating a waits-for relationship between the threads. 3) The dependency between τ_h and τ_l causes priority inheritance in the scheduler. Due to this, the scheduler executes τ_l which eventually returns from the function it executes in the component, notes the set contended bit, and calls the stack manager to notify it that a stack is available for use. Thus, the stack manager invokes the scheduler to wake τ_h. 4) τ_h wakes, returns to the component, and acquires the stack. Although this may seem complicated, a motivation behind component-based systems is the separation of concerns. Both the stack manager and scheduler are configurable, and are implemented at user-level in separate protection domains.

This example involves only a single stack, and two threads. When more stacks are allocated to a component, the overheads of invoking the stack manager and performing the priority inheritance will only occur when all of the stacks are in use. In the current implementation, the thread that inherits the priority of τ_h is arbitrarily chosen amongst those holding stacks which does not effect the theoretical bounds as PIP must already consider chained blocking. This mimics multi-unit resource protocols as described in [6].

Multi-unit priority ceiling protocol in COMPOSITE. In contrast to PIP, PCP takes active action to avoid *possible* contention even when there is no actual contention. Specifically, even when there is not resource contention, the system ceiling is updated. This ceiling must be maintained in a central location so that it can be updated whenever a stack in *any* component is requested, thus in COMPOSITE, the stack manager maintains the ceiling. Consequently, whenever a thread wishes to use a stack, or release a stack, it must invoke the stack manager. This is analogous to user-level mutexes that implement PCP in Linux: a system call to the kernel must be made to update the ceiling, even when there is no contention.

Figure 2(c) depicts an invocation using PCP *without* contention. 1) τ invokes the component, and immediately calls the stack manager to update the ceiling. 2) τ returns to the component and acquires the stack. 3) After the execution is complete, τ calls the stack manager to lower the system ceiling. 4) τ releases the stack and returns to the invoking component. When there is contention, PCP takes actions almost identical to those in Figure 2(b) except that τ_l first raises the system ceiling before initially taking the stack.

Multi-unit PCP [6] is used in COMPOSITE to maintain the guarantee that any thread will only experience contention once during its execution, thus limiting blocking time to the maximum hold time for one stack. Importantly, this means that the highest priority thread holding a stack in a component should be the one that inherits a suspending thread's priority.

Coordination with resource sharing for other resources. Although this paper focuses on the protocols, analytical models, and resource allocation policies for run-time stacks, COMPOSITE also includes resource sharing protocols for more traditional semaphore protected resources [16]. These policies are implemented in a lock component. If both the lock component, and the stack manager provide PIP, the waits-for graph that takes both types of resources into account is maintained by the scheduler component. Specifically, when a thread is suspended by the stack manager, or the lock component, the scheduler is informed of the *dependency* formed between this thread

[2]Though atomic instructions could be used, COMPOSITE uses a "restartable atomic sequence" [15] for atomic removal of a stack from the freelist.

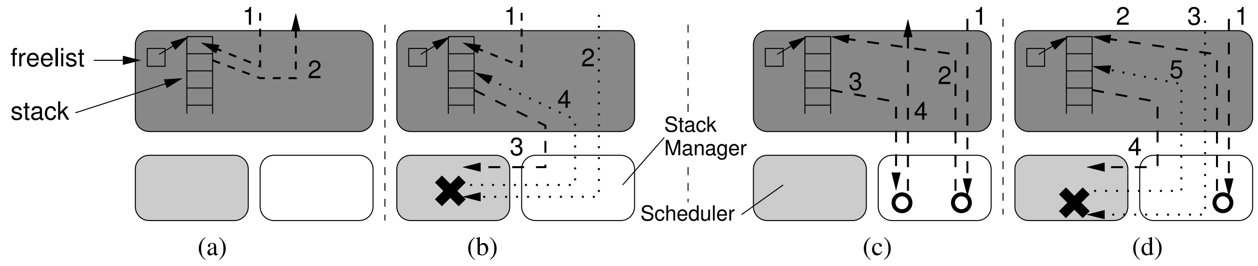

Fig. 2. The dashed line is a low-priority thread, while the dotted line is a high-priority thread. Four situations each depict a component with managed stacks (dark), a stack manager (white), and the scheduler (light). An **X** in the scheduler and an **O** in the stack manager designate a thread blocking, and ceiling manipulations, respectively. Priority inheritance without contention (a), and with (b). Priority ceiling without contention (c), and with (d).

and the one it that should inherit its priority. The scheduler uses a mechanism similar to shadow tasks in Shark [17] to provide proper priority inheritance that integrates all different types of resources. If all resource types use PCP, a single centralized ceiling will need to be maintained either in the lock component, or in the stack manager.

Managing stacks for the stack manager and scheduler. Who manages the stacks for the stack manager and the scheduler? We take a practical approach to answering this question: we pre-allocate enough stacks to satisfy all invocations in these components (*i.e.* the stack freelists contain a number of stacks equal to the total number of system threads). These components have simple code-paths and don't require much stack space, so we believe this is an acceptable design. Importantly, although we allocate stacks for all threads in these components, this enables the schedulability-aware allocation, and contention management of stacks for *all other* components.

Comparison to the stack resource protocol. The Stack Resource Protocol (SRP) [18] is a common resource sharing protocol with theoretical properties similar to PCP. SRP, as PCP, is not optimistic in the sense that the system ceiling must be altered on each component invocation. In SRP, the ceiling must be tightly integrated into the scheduling policies of the system to ensure that thread preemptions only happen when the priority of the preempting thread is greater than the system ceiling. As the scheduler must be invoked even when there is no contention (*i.e.* for ceiling maintenance), the practical overheads due to component invocations are similar to those for PCP. In COMPOSITE, the scheduler is implemented as a separate user-level component, thus requiring inter-component communication when it is involved. Though component invocations in COMPOSITE are efficient (similar to optimized microkernels [2] and system call overheads in monolithic systems), this cost is significant compared to the uncontended priority inheritance operations. Even in a monolithic system such as Linux, operations that require scheduler interaction (ceiling maintenance) require relatively expensive system calls compared to optimized PIP paths [13] that do not. Given the similar analytical bounds for PCP and SRP, and the comparable practical overheads, we focus on comparing PCP and PIP. We leave a more in-depth study of SRP for future work.

III. Component-Based System Model

$\{\tau_i, \ldots\} \in T$ is the set of tasks. We assume tasks are implemented as threads, and we use the terms interchangeably henceforth. We assume a simple periodic task model in which each task consists of an infinite number of jobs. Each job activates at the thread τ_i's periodicity, p_i, executes for a maximum execution time of e_i, and has a deadline equivalent to the activation time of its next job. We assume fixed priority scheduling with unique priority assignments. The set of threads with higher-priority than τ_i is hp_i.

$\{c^x, \ldots\} \in C$ is the set of system components, that comprise the executable system[3]. Each component depends on a set of other component's functionality that it accessed through their functional interface via component invocation. This *dependency* relation is captured by d^x that contains all components that are depended on by c^x. We describe the set of all components transitively depended on by c^x as $D^x = d^x \cup (\bigcup_{c^y \in d^x} D^y)$.

The system is additionally described by:

- A τ_i executing in c^x can *invoke* the components in d^x. We assume that invocations are synchronous: When a thread invokes $c^y \in d^x$, it will continue execution *in* c^y, and will resume execution in c^x only when it *returns* from c^y. Many μ-kernels [1], [7], [8] and component-based systems [10], including COMPOSITE, are based on synchronous IPC. In this way, invocations mimic function calls, but operate on components rather than functions. Invocations have non-zero execution overheads, in the worst case characterized by O.
- When a thread invokes c^x, the worst-case processing done in that component is e^x. When a component is invoked, the worst-case number of invocations it makes to $c^y \in d^x$ is $v^{x,y}$. In keeping with a component model, both e^x and $v^{x,y}$ are specified independently for each component. Although $v^{x,y}$ references two components, it is purely a function of c^x. Its implementation could change to invoke c^y more or less, independent of c^y.
- Each thread begins execution in a specific component (*i.e.* with the thread's `main`). $h_i \in C$ is τ_i's "home" component.

[3]To make the notation easier to follow, we use subscripts to denote the selection of a specific thread, and superscripts to denote selection of a component.

We simply write h when the specified thread is unambiguous. Given thread's home components, the set of threads that can invoke a specific component is defined as $t^x = \{\tau_i | c^x \in D^{h_i}\}$. We define $lp_i^x \in t^x$ to be the set of threads with lower-priority than τ_i in c^x.

- A component, c^x, contains a total of s^x number of execution stacks. When a specific stack is used as an execution context for a thread, it cannot be used by another thread, *i.e.* it is a resource shared in a mutually exclusive manner between all threads that can invoke c^x. The number of stacks that aren't being used for any thread's execution at a point in time (*i.e.* the number that are available) is $0 \leq a^x \leq s^x$. When a thread invokes a component (c^x) and there are no available stacks, $a^x = 0$, that thread must block waiting for a stack to become available. The overhead of this operation (including blocking the thread, inheriting priority, and context switching twice) is M, and we call this operation a *stack miss*. To avoid unbounded priority inversion, resource sharing protocols are used. The maximum amount of time that a thread τ_i can block waiting to use a stack in component c^x is denoted b_i^x. The total maximum blocking time for τ_i is b_i.

The stack used for a thread in its home component is not counted in s^h as it is never shared between threads. Unless a thread terminates (*i.e.* returns from `main`), it will never relinquish this stack.

The worst-case execution time of a thread, e_i, if executed alone on the system is determined by the worst case execution times of the individual components it invokes, and the number of invocations made between them. Thus, it can be determined from the independent parameters of the components used in the system: $e_i = e^{h_i}$.

$$e^x = E^x + I^x \tag{1}$$

where the cost of component execution and the cost of invocations is

$$E^x = e^x + \sum_{\forall c^y \in d^x} v^{x,y} E^y, \quad I^x = \sum_{\forall c^y \in d^x} v^{x,y}(O + I^y)$$

Assumption: known, acyclic component graph. A graph of components that define a system of hard real-time tasks is loaded as a single unit. A system specification includes all components, and their dependencies. The system ensures at run-time that communication is only made between components as described in the specification using capabilities [19]. Given this specification, a number of static analysis are performed on the system. (1) The priority ceiling protocol requires that the maximum priority of any task that can invoke a specific component is known a-priori. This value can be ascertained solely from the tasks that exist in the system, and the graph structure. Specifically, the ceiling for c^x is the maximum priority of any thread, $\max_{\forall \tau_i | c^x \in D^{h_i}}(prio(\tau_i))$ where a thread's priority is $prio(\tau_i)$. (2) Note that e_i given the definition above is not bounded if cyclic component graphs are allowed. Thus one analysis that is performed is to ensure that the graph is acyclic. To bound e_i, we ensure at load-time that the component graph is acyclic, i.e. $c^x \notin D^x$. In addition to making the analysis possible, cyclic structures in component based systems (and μ-kernels) are dangerous as they can lead to deadlock when used with synchronous IPC. Although PIP does not avert deadlock, this assumption removes any cyclical-wait relationships between threads and shared stacks, thus *preventing stack-based deadlock* in COMPOSITE.

The generality of this model. Although it seems that certain parameters of this model are specific to COMPOSITE– such as the number of stacks per component, or the acyclic nature of the component graph – we argue that the model captures general characteristics shared by many systems. Systems that distribute execution contexts, and uses a synchronous model of communication between them, can be described in this model. For example, systems that use synchronous IPC between threads [1] must answer the question of how many threads to serve requests in an invoked component.

IV. TASK BLOCKING TIME CALCULATION

The maximum resource hold time is the maximum amount of time between when a task acquires and releases a resource. It is common to assume that this value is known, or is easily derived from individual resource hold times. However, as we will see, allocating stacks to different components has the effect of changing the stack hold times. Thus instead of assuming the maximum stack hold times for different stacks are constant, we calculate them from the structure of the system. We assume that high-priority threads do not self-suspend while holding resources (stacks).

A. Priority Ceiling Protocol Blocking Time Calculation

For PCP, the maximum blocking time for a thread τ_i is

$$b_i = \max_{\forall c^x \in d^h}(b_i^x(PCP)) \text{ where } b_i^x(PCP) = e^x + M \tag{2}$$

Blocking time refinement: stack allocations. This analysis is pessimistic: it doesn't consider either thread priority, or stacks allocations. τ_i will not block in c^x if stacks are available. In the worst case, we can guarantee that a thread will not block when invoking a component if it is amongst the s^x lowest (inherited) priority threads that can invoke c^x.

$$b_i^x(PCP) = \begin{cases} \max_{\forall c^y \in d^x}(b_i^y) & \text{if } |lp_i^x| < s^x \\ e^x + M & \text{otherwise} \end{cases} \tag{3}$$

When we refer to b_i^y, assume that we mean the textually closest definition of b_i^y (*i.e.* $b_i^y(PCP)$).

B. Priority Inheritance Protocol Blocking Time Calculation

We expand on Equation 2 to define the blocking time for PIP. In this case every invocation can result in resource contention, thus the maximum blocking time for a thread τ_i is

$$b_i = \sum_{\forall c^x \in d^h} b_i^x(PIP) \text{ where } b_i^x(PIP) = e^x + M + \sum_{\forall c^y \in d^x} b_i^y$$

This assumes that when τ_i invokes c^x, $a^x = 0$, thus causing the thread to block waiting for the stack to become

available. The thread holding the stack executes, and when it invokes other components (in d^x) they also have no stacks available. For each invocation, there is contention on the resource, thus causing priority inheritance. This is equivalent to priority inheritance's well-known "chained blocking" [5]. Thus a thread's block time is dependent not only on the execution time of the thread holding the stack (e^x), but also on its blocking time for other components as well.

Blocking time refinement: stack allocations. Similar to Equation 3, we refine this analysis by taking into account thread priority and stack allocations.

$$b_i^x(PIP) = \begin{cases} \beta_i^x & \text{if } |lp_i^x| < s^x \\ e^x + M + \sum_{\forall c^y \in d^x} b_i^y & \text{otherwise} \end{cases}$$

$$\text{where } \beta_i^x = \sum_{\forall c^y \in d^x} b_i^y$$

Threads only experience stack misses when their initial priority places them in the set of lower priority threads of size greater than s^x. In effect, the s^x lowest priority threads will not block in contention for stacks. It should be noted that they *will* contend for stacks when they inherit a higher-priority, but in such cases, their blocking time will not be increased, only the blocking time of the higher-priority thread (τ_i in this case).

This analysis is pessimistic for two reasons. First, execution at the priority of a thread can only suffer from "chained blocking" once per component, thus limiting the total number of blocking occurrences. Second, the number of lower-priority threads also limits the impact of chained blocking events. We refine the analysis based on these two observations.

Blocking time refinement: number of components. We observe that "chained blocking" can only occur a maximum of $|D^{h_i}|$ times for τ_i. If τ_l – a low-priority thread – holds a stack in c^x, and a higher-priority thread, τ_h, blocks waiting for the stack, when τ_l completes its execution and frees the stack, all components it invokes (in D^x) will have an available stack. This stack will be used without contention by τ_h, or by any thread that blocks τ_h and inherits its priority. Thus, once a stack miss occurs in c^x while executing with τ_h's priority, threads invoking c^x will *not suffer another stack miss*. We can bound the number of blocking occurrences by the number of components τ_h can invoke (*i.e.* $|D^{h_i}|$). A thread will not block if we can guarantee there are enough stacks for it in a given component.

$$b_i = \sum_{\forall c^x \in D^{h_i}} \begin{cases} 0 & \text{if } |lp_i^x| < s^x \\ e^x + M & \text{otherwise} \end{cases} \quad (4)$$

V. Component Stack Allocation and Schedulability

If we assume that $\forall x, s^x = 1$, then we can solve the blocking terms for each thread. We use response time analysis [20] to determine if the system is schedulable. We use the fixed point construction in [21]:

$$R_i^{t+1} = e_i + b_i + \sum_{\forall \tau_j \in hp_i} \left\lceil \frac{R_i^t}{p_j} \right\rceil e_j \quad (5)$$

hp_i returns the set of threads that have a higher priority than τ_i. $R_i^0 = 0$, and the solution iterates until either $R_i^{t+1} = R_i^t$, or $R_i^t > p_i$, in which case the response time of the task is greater than its deadline and it is not schedulable.

Stack allocation algorithms. If we assume that, given the blocking factors (Equations 3 and 4) and thread execution times, the analysis in Equation 5 shows the system is not schedulable, we observe that increasing the stack allocation to components ($s^x > 1$) decreases the blocking factors. We assume that the system is schedulable if $\forall x, s^x = |T|$, which removes all blocking terms[4].

We propose two greedy heuristics, one for allocating stacks when using PCP and one for PIP. They attempt to make the system schedulable while allocating the fewest number of stacks. Both algorithms, take the set of all threads, $F \subseteq T$, that *fail* to satisfy the response-time analysis of Equation 5.

A. Stack allocation for PCP

We observe that the only way to decrease the blocking time for threads when using PCP, is to add stacks to components that can block the thread, but not those invoked via dependencies from these. The function `max_component` finds this set. Algorithm 1 uses this function to find the component, c^x, of this set that has the maximum execution time – the one contributing to the thread's blocking time as shown in Equation 3. It will do this for the highest priority thread missing deadlines, τ_h. If stacks are allocated to a component for this thread, it will also benefit other threads in F dependent on the same component. Stacks are allocated to prevent τ_h from blocking on stacks for c^x. Once stacks are added, the response time analysis of those threads in t^x is recomputed. If any of them now pass, they are not considered further, otherwise the algorithm repeats.

Algorithm 1: PCP Stack Allocation

```
   Input: F: Set of threads that fail the response-time analysis
1  while F ≠ ∅ do          // While threads miss deadlines
2      τ_i = get_highest_prio_thd(F)
3      c^m = max_component(F, τ_i)
       // Increase the number of stacks in c^m
4      s^m = |lp_i^m| + 1
       // t^m is all threads that can invoke c^m
5      for τ_i ∈ t^m ∩ F do    // Thds w/ new blocking times
6          if pass_response_time_analysis(τ_i) then
               // remove the thread from F
7              F = F\τ_i
8      end
9  end
10 return
```

$$\texttt{max_component}(F, \tau_i) = \max_{\forall c^x \in \texttt{can_blk}(h_i, \tau_i)} (e^x) \text{ where}$$

[4]If this isn't true, then the system must be fundamentally redesigned by changing the tasks, execution times, or processing power.

$$\texttt{can_blk}(c^y, \tau_i) = \bigcup_{c^z \in d^y} \begin{cases} \texttt{can_blk}(c^z, \tau_i) & \text{if } |lp_i^z| < s^z \\ c^z & \text{otherwise} \end{cases}$$

Algorithmic Complexity. The complexity of algorithm 1 is $O(|C|(|T|log|T|) + |C|(|E||C| + |T|RTA))$ where RTA is the cost of conducting a response time analysis. Although much work has been done to improve the efficiency of this operation [22], in practice we have observed that this term dominates the cost of the algorithm.

This optimization is run *offline*, and we find the runtime to not be prohibitive (average runs for a Java implementation take on the order of two seconds for systems with 1K threads on a Pentium 4 system running at 2.8 Ghz).

B. Stack Allocation for PIP

Stack allocations to *any* c^x can decrease the blocking time of threads in t^x, and components with a higher e^x will decrease thread blocking time by a larger amount. The PIP stack assignment algorithm uses these relations to allocate stacks toward schedulability. The PIP algorithm is identical to Algorithm 1 except that `max_component` is defined differently. `max_component` instead finds the component that has the best ratio of decreased blocking time for threads that miss their deadlines to the number of stacks required for this decrease in blocking time. As in Algorithm1, stacks are allocated to these components until no threads miss their deadlines.

$$\texttt{max_component}(F, \tau_i) = \max_{\forall c^x \in D^{h_i}} ((e^x \times |t^x \cap F|)/|lp_\beta^x|)$$

Algorithmic Complexity. Only the `max_component` function differs in complexity from the stack allocation algorithm for PCP. The complexity of that procedure is $O(|C||T|)$ ($|T|$ to compute the intersection). Thus the final complexity is $O(|C|(|T|log|T|) + |C|(|C||T| + |T|RTA))$. The algorithm is run *offline* and we have not observed it to take more than three seconds for systems with 1K threads, we have not found the cost to be prohibitive (on the same system as in Section V-A).

C. Considering Different Component Stack Requirements

The presented algorithms assume that the memory usage per stack is the same for different components. This assumption has a practical motivation: it is common for thread packages to allocate stacks for threads of a fixed size. However, if maximum stack usage could be profiled or statically determined, then per-stack memory allocations would differ between components. We save an evaluation of this for future work. However, we note that the algorithms that take a thread that cannot meet deadlines and choose a component to receive stacks, can be enhanced to consider the amount of memory required to do so.

VI. Experimental Evaluation

In this section, we combine empirical measurements from Composite with simulation results to analyze the effectiveness of (1) PCP and PIP for different system configurations, and (2) the capabilities of the algorithms in Section V to make the system schedulable while minimizing the memory required for stacks.

Operation	Average	Stddev	Worst Case
context switch	1.42	0.06	29.22
O^{pip}	0.56	0.1	6.05
O^{pcp}	1.53	0.04	18.17*
M	7.39	0.09	76.63*

TABLE I

Composite microbenchmarks (measured in μ-seconds). Context switches include a single invocation to the scheduler. Values marked with * are qualified in the text.

A. Composite *Performance Characteristics*

These experiments are run on an Intel Core 2 Duo with only one core enabled running at 2.33 Ghz. Composite is loaded using Hijack techniques [23], and it uses only the Linux timer drivers. We measure all operations using the `rdtsc` cycle counter provided by the architecture.

Although we intend to study generic component-based system configurations, we will particularly study a system configuration motivated by Composite. Thus, we empirically measure or infer the costs of key operations in Composite: the maximum invocation overheads for both PIP and PCP (we will denote these as O^{pip} and O^{pcp}, respectively), and the maximum cost of a stack miss, M.

In measuring the worst-case costs of these different operations, we wish to avoid interference that results in measurement of asynchronous events such as interrupts. Specifically, measuring the worst-case execution time of a fast operation will be dominated by the processing of the timer interrupt if it preempts the tested code. Interrupt processing time must be taken into account in a real-time system, but it must be taken into account *separately* from the worst-case execution times of unrelated operations.

Often one can disable interrupts to obtain accurate measurements. However, as we wish to measure events that involve both kernel and user-level, the natural transitions for component invocation will re-enable interrupts (*i.e.* the kernel ensures that interrupts are enables whenever switching to user-level). Instead of changing this kernel behavior, we measure the key operations carefully and avoid interrupt interference: we set up the task conducting the measurements as a real-time task with a periodicity of a single timer-tick. At the beginning of the task's execution, it manually writes to each cache-line in a large array to flush the cache's content. Next, the operation we wish to measure is performed, and the timing recorded. This is done every clock tick on an otherwise quiescent system till we record 1K readings. We found this method to work well for somewhat atomic operations, such as measuring the maximum context switch latency, or the O for PIP. However for operations such as O^{pcp} and M, this is insufficient. We found the worst-case reported measurement included the worst case cost for a *single* invocation or context switch, but that the rest of the invocations and other operations took times closer to their average.

To compensate for inaccuracy of our worst-case measurements, we observed that both O^{pcp} and M are composite operations of primitive operations for which we can accurately

measure the worst case. Thus O^{pcp} and M worst cases are synthesized from the worst-case measurements of their constituent operations. The average and standard deviations of all measurements are directly derived from the measurements. Table I shows the costs of these operations in COMPOSITE. In this study, our goal is to achieve some understanding of the *relative* costs of these operations (especially O^{pip} and O^{pcp}).

B. System Generation and Simulation

To study the schedulability properties of the different resource sharing protocols and stack allocation algorithms, we generate component graphs, and task sets, and apply our analysis to them. Although task set generation is common and well-studied, synthesizing a component hierarchy is not. Our goal is to enable the generation of systems that are parameterized in such a manner that they can capture a wide variety of component-based and μ-kernel systems. Important differences between these systems are (1) the aspect ratio of the graph, or how "deep" the system is (*i.e.* the maximum number of nested invocations) versus how many components are in the system, (2) the ratio of invocation costs to in-component execution, (3) the number of dependencies that components have, (4) and the maximum number of invocations that are made between components.

Component graph generation. We wish to show which types of graphs and task sets the different resource sharing protocols are superior for, and what types of systems can benefit most from stack allocations. Thus, we generate random component graphs parameterized by the tuple $(\delta^{max}, w^{avg}, i^{ratio}, d^{avg}, n^{avg})$. δ^{max} is the maximum depth of the system, while w^{avg} is the average width of a system. Although width is a synthetic parameter (i.e. real systems do not have a defined width), it, along with δ^{max}, enables the model to control the ratio of the maximum number of nested invocations to the number of components. For example, a monolithic kernel such as UNIX could be seen as a very shallow component graph ($\delta^{max} = 2$, with large w^{avg} [5]), while a pipeline of processes (*e.g.* a UNIX pipeline – where processes have execution dependencies) has a small width and is deep. i^{ratio} is the ratio of the average e^x of components in the system to the cost of an invocation (O^{pip}). d^{avg} is the average number of dependencies a component has, while n^{avg} is the average maximum number of invocations made over a dependency each time a component is invoked. All random variables use an exponential distribution.

A component graph is generated by creating a "grid" of components, and then connecting them via dependencies. The grid has a depth δ^{max}, and at each level (each row), we generate on average w^{avg} components. Each component at level l has dependencies on d^{avg} components in levels $[l+1, \delta^{max}]$. Each of the invocation "edges" signifies a maximum of n^{avg}, on average, invocations whenever the component is invoked. For some value of O^{pip}, each component is assigned an e^x that is on average $i^{ratio}(O^{pip})$. If system graphs are created that are disconnected, they are discarded.

[5]Any process-driven kernel [24] with a kernel stack for each thread, including Linux, could be seen as a system where the "kernel component" has $s^{kernel} = |T|$.

Task set generation. Given a component graph, we generate task sets to analyze their schedulability. $|T|$ tasks are randomly assigned home components. We favor placing threads in lesser levels to ensure that the "functionality" of the system defined by higher levels is used. Thus, tasks are assigned a random home component at level l where l is exponentially distributed with an average of 1.2 (where the top level is 0). The thread's e_i is generated using Equation 1. We generate task sets with a randomly distributed target utilization. Given a utilization target, $0 < U \leq 1$, we assign each thread a p_i using a random variable with an exponential distribution and an average of $U/|T|$. Tasks are assigned priorities based on rate-monotonic assignment: tasks with lower p_i have higher priority.

C. Schedulability of PCP and PIP

To evaluate the properties of component-based systems implementing either PCP or PIP, we study the effects of altering system parameters to determine which significantly affect the system schedulability. In doing so, we compare the schedulability properties of PIP and PCP. Although PCP is analytically superior to PIP in the literature as it avoids chained blocking, system implementations of PCP as outlined in this paper can have more overhead as they require *active* maintenance of the priority ceiling. In COMPOSITE, this requires invoking the stack manager once to raise the ceiling, and once to lower it, on every component invocation. In contrast, PIP does not require priority/ceiling manipulations in the uncontended case. This real-world system overhead requires an analysis of the factors that are beneficial or detrimental to PCP and to PIP.

We generate system configurations as described in Section VI-B with the following default parameters: δ^{max} = 10, w^{avg} = 4, i^{ratio} = 10, d^{avg} = 2, and n^{avg} = 1.2. We choose O^{pip}, O^{pcp}, and M based on the actual results from COMPOSITE in Section VI-A (note that to compare PCP and PIP, the ratio of these results and the ratio of execution time to the these overhead (i^{ratio}) are salient, not the absolute values). For the task model, by default we choose $|T|$ = 8.

To compare PIP and PCP, we generate graphs with the total utilization ($U = \sum_{\tau_i \in T} e_i/p_i$) of the system uniformly at random in the range $(0, 1]$. We generate a graph, calculate task execution times assuming that there is no invocation overhead ($O = 0$), and no blocking times (*i.e.* that each component has $|T|$ stacks). We discard any generated systems that don't pass a response time analysis under these conditions. Thus systems are generated for a system without any overheads from stack contention. Using the same component graph and task set, we add in the actual resource sharing overheads (O^{pip}, O^{pcp}, and the calculated b_i) for both $s^x = 1$ (maximum blocking overheads) and $s^x = |T|$ (minimal blocking overheads). This enables a comparison of the two resource sharing protocols and associated system overheads. Unless otherwise noted, we generate 1000 graphs that are schedulable with $O = 0$ and

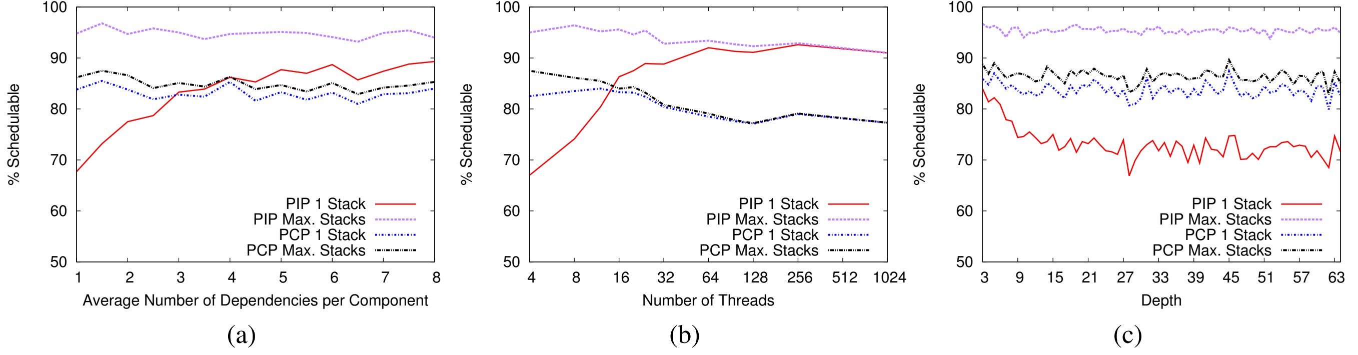

Fig. 3. Percent of randomly generated systems with $O = 0, s^x = |T|$ that are schedulable, varying parameters.

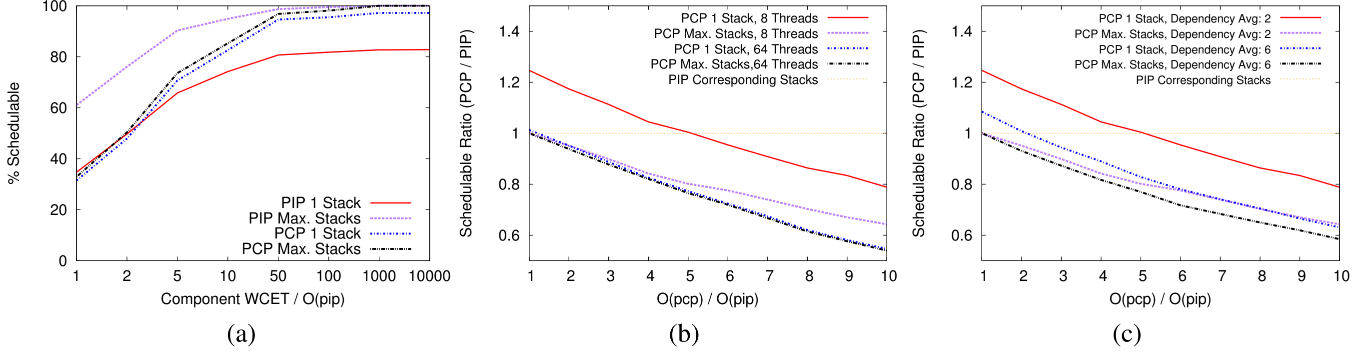

Fig. 4. (a) Modifying the ratio of in-component execution time, to the cost of O^{pip}. (b)-(c) compares the schedulability of PCP and PIP when changing the ratio of O^{pcp} to O^{pip}.

$s^x = |T|$, and report the percent of them that are schedulable using PIP and PCP.

Figure 3 shows the percent of the generated graphs that are schedulable for both PCP and PIP when each component has one stack, or a maximum number of stacks. Note that the y-axis shows only the upper 50%. We keep all variables constant at default values, and change one control variable. To study which properties of the component graph affect the schedulability, we vary d^{avg} in (a), the number of threads, $|T|$ in (b), and the depth of the generated graph, δ^{max} in (c) (note that because we are changing the depth, and keeping the average width constant, this alters the aspect ratio of the graph). Although we do not plot the results, we vary n^{avg} (the average maximum number of invocations on a dependency), and find that the results and trends are very similar to the plots for d^{avg}, thus for brevity we omit them. We will discuss the similarity at the end of the section. In Figure 4(a), we study the effect of the ratio of the execution time done in each component to the cost of invocations (*i.e.* $i^{ratio} = e^x/O^{pip}$). In this case, the x-axis is the ratio of component execution time to invocations for PIP (note the change in the y-axis). PCP has a correspondingly higher invocation overhead that is taken into account in the plot.

In these results, we use a ratio of O^{pip} to O^{pcp} derived from measurements in COMPOSITE. However, other systems might have different invocation overheads depending on how they implement PIP and PCP. We study the effect of varying O^{pcp}/O^{pip} on the schedulability of systems under PCP and PIP in Figure 4(b) and (c). We plot the ratio of the percent of schedulable systems for PCP to PIP. The plot is normalized to PIP, so we only plot PCP values. Values above y=1 mean that for that ratio of invocation costs, PCP can schedule more systems, while values below y=1 means that PIP can schedule more. We choose to vary the number of threads ($|T|$) in (b), and the number of dependencies (d^{avg}) in (c) as these are the variables deducted from Figures 3(a)-(c) for which PIP and PCP diverge most significantly in behavior. We see in Figure 4(b) that when both PIP and PCP have components with only one stack, when the ratio of their invocation costs is less than 5, PCP schedules more systems due to the large blocking times induced by PIP. However, for most other system settings, notably when there are many system threads, or the costs of invocations is significant, PIP can schedule more systems. In Figure 4(c), we vary the number of dependencies, and find that smaller dependency numbers hurt PIP (see Figure 3(a)), thus relatively increasing PCP's schedulability.

Discussion. Given this data, we make a number of observations: 1) PIP has a lower overhead than PCP for normal invocations, thus when each component is allocated $|T|$ stacks (eliminating blocking overhead), PIP performs better than PCP across all tests. 2) There are a large number of systems that are not schedulable using PIP without stacks (*e.g.* $> 30\%$ of the systems in Figure 3(a) with 1 dependency). For these cases, intelligent stack allocation can benefit the system. 3) The overhead of invocations for PCP relative to PIP has a significant effect on the comparison between the two. However, when stacks are available, PIP exceeds PCP (Figure 4(b) and (c), `Max. Stacks` plots). 4) Trends for PIP and $s^x = 1$ are

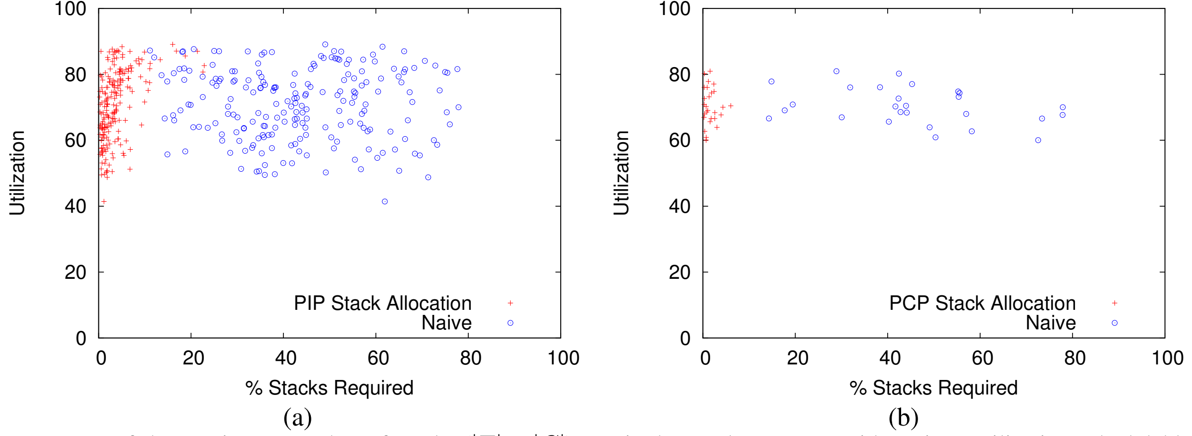

Fig. 5. Percentage of the maximum number of stacks ($|T| \times |C|$) required to make systems with a given utilization schedulable that were not schedulable with $\forall c^x, s^x = 1$, but are with $s^x = |T|$. (a) Plots for the 207 (of 1K) systems that did not pass for PIP, while (b) plots the 27 (of 1K) systems for PCP. A vertical split in either graph yields the number of systems (on the left of the split) that will be schedulable given a specific percent of stacks.

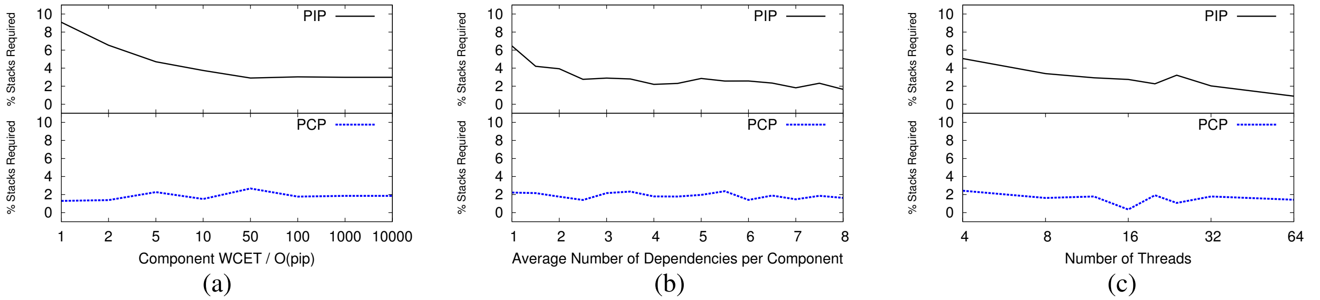

Fig. 6. The percent of $|T| \times |C|$ stacks required for systems using PIP and PCP as we vary the system variables that Section VI-C showed affect schedulability. (a) Varying the ratio of component execution time to invocation overhead. (b) Varying the average number of dependencies a component has. (c) Varying the number of tasks. We plot PIP and PCP separately as the number of systems that can benefit from stack allocation differs for each. Thus the percent of stacks required is averaged over a different number of systems for each.

largely dependent on the ratio between the blocking term, and the thread's p_i. The number of threads increases a thread's p_i if the system utilization is held constant, but the b_i does not comparably increase. Thus, more threads make the blocking term contribute less to the response time analysis. Comparably, d^{avg} also affects this ratio. As d^{avg} grows, the execution time assuredly grows, but b_i doesn't necessarily as the additional connections might be to components that are already counted in the thread's blocking term. 5) When $|D^h|$ is closer to 1, the blocking terms for PCP and PIP are closest. When δ^{max} is close to 1, so is the dependency set. Thus, for systems with a short depth, PIP (with $s^x = 1$) is closer to PCP, as shown in Figure 3(c). 6) Figures 4(a)-(c) confirm our intuition: as the invocation overhead becomes inconsequential ($e^x >> O$ or $O^{pcp} \approx O^{pip}$), more systems become schedulable (a), and the blocking term matters more. When both the blocking term and invocation costs diminish, the schedulability approaches optimal (a, PIP, $s^x = |T|$).

System design guidelines. We derive from the data a number of design guidelines for systems that must choose between using PIP and PCP for stack sharing. When $s^x = 1$, PCP is often superior to PIP. However a number of factors can change this: for larger numbers of threads, larger numbers of component dependencies, and small ratios of e^x/O, PIP can schedule more systems. Practical system implementation details greatly affect the schedulability of PCP vs. PIP. However, when stacks are available, the blocking term of PIP disappears, and its performance (when there is *any* overhead for normal invocations for PCP) is superior to PCP. However, requiring that $s^x = |T|$, thus that the system devote memory for $|T| \times |C|$ stacks is unrealistic in many systems, especially embedded systems. In the next section, we investigate how stacks can intelligently be allocated to minimize used memory, while ensuring schedulability.

D. *Stack Allocation for Schedulability*

Results in Section VI-C show that both PCP and PIP benefit from having additional stacks allocated into components. The blocking terms in PCP are smaller than in PIP, so PIP tends to benefit more from having these overheads removed. In this section, we study the effectiveness of the algorithms described in Section V to make systems that are not schedulable with $\forall c^x, s^x = 1$, schedulable while attempting to minimize the number of stacks allocated. In this section, we report on the amount of stacks required to make these systems schedulable.

Figure 5 plots the number of stacks required for systems that were not schedulable with $s^x = 1$ and the utilization of

that system. We plot two possible stack allocation methods for both PIP and PCP and show what percentage of the total possible allocated stacks ($|T| \times |C|$) are required. The *naive* method takes into account the fact that in COMPOSITE, we can generate t^x (where $t^x \leq |T|$) for each component as the system knows the dependency sets of each component, and each thread's home component. This method gives each component a number of stacks such that $s^x = |t^x|$. The second method we explore is the use of the intelligent algorithms from Section V. In the graphs, this is labeled *stack allocation*.

Figures 6(a)-(c) depict the percent of the total number of stacks required to make the system schedulable for systems that aren't schedulable with $s^x = 1$, but are with $s^x = |T|$ while varying system variables. This effectively compares against the case where a system does not intelligently manage stacks, and does not understand the communication behaviors of components. We study the effect of system characteristics on the effectiveness of the algorithms of Section V. For each of the points, we generate 1K systems that are schedulable when $O = 0$ and $s^x = |T|$. Of those systems that can benefit from stack allocation for PIP and PCP (separately), we plot the average number of stacks required to make the system schedulable. We plot these separately for PIP and PCP as the number of systems, and the actual system configurations used, differ for the protocols.

Discussion. In Section VI-C, we conclude that PIP has potential to provide a high-degree of schedulability for many systems, but that significant memory is required for stacks to do so. In this section, we study how much memory is practically required using intelligent stack allocation. Although system configuration parameters do affect the number of required stacks for PIP, for all configurations on average the system never requires more than 10% of the maximum number. We believe this makes the usage of PIP when paired with intelligent stack allocation for schedulability a practical option for many systems.

Protocol	Average Stack Memory Used	and Saved
PIP	46.2 KB (3.7%)	1188 KB
PCP	17.89 KB (1.5%)	1167.8 KB

TABLE II
MEMORY USED FOR STACKS FROM FIGURE 5.

We summarize the results for the stack allocation algorithms in Table II. We assume that each stack requires 4KBs of memory (this is the value taken from COMPOSITE– it will vary for different systems). We show how much memory (on average) is used for stacks, and how much is saved compared to allocating $|T|$ stacks to each component (*i.e.* $|T| \times |C|$ stacks). The value in parenthesis denotes the percent of the memory required for $s^x = |T|$ that is used to make the system schedulable.

VII. RELATED WORK

Systems that break execution up into different components, each separated via memory isolation (*e.g.* hardware page-tables, or software memory safety), separate the execution contexts of the system into the different components. We focus on systems with synchronous communication between components. Many systems exist that implement this model, each with different mechanisms for locating execution contexts. LRPC [25] dispatches an invocation to an execution context identified by the kernel, while protected procedure calls in K42 [26] acquire execution contexts in the invoked component in user-space (as in COMPOSITE). The L4 [1] family of μ-kernels use synchronous IPC between threads. Specific implementations use the separate threads primarily to identify the execution contexts [8], [7], and priority inheritance is used to resolve contention on concurrently required threads. Middleware systems intelligently manage thread pools for predictability [4]. A natural question is "how many threads should be in the pool?" This paper helps answer this question. All of these systems further motivate this work by raising the question of how execution context (stack) sharing affects schedulability. Notably, we do not know of any other systems than the one presented that have implemented PCP in the invocation path.

Pebble [3] is another component-based system that associates a set of stacks with a thread, rather than with specific components. When an invocation on a component is made, on of these stacks is mapped into the component for thread execution. We study the stack allocation to components instead of specific threads for a variety of reasons. (1) Performance: page table manipulations (to map and unmap the stack) has significant overhead compared to the otherwise highly optimized invocation path. (2) Configurability: to find a free address to map stacks into a component dynamically, the kernel must manage the virtual address space of components (*i.e.* allocate/free virtual ranges). This complicates the kernel implementation, and the ability of user-level components to manage their own virtual address spaces. A goal of component-based systems is to define all such policies as user-level components. (3) Security: a thread's stack can map into a variety of components when they are invoked. This can allow one component to observe the stack state (local variables) of another component, causing information leakage. In studying stack allocation to components, we solve a problem that has broader applicability to middleware, microkernels, and other such systems that use thread-pools.

Although in this paper we compare the overheads of PCP and PIP in relation to stack contention, PCP also causes overhead for user-level lock implementations that seek to avoid system calls in the uncontended case [16], [13]. Thus, elements of this work are of general concern: PIP's optimistic implementation allows user-level, efficient implementations of uncontended lock access similar to the fast PIP path in our system. In contrast, PCP requires kernel system-call invocations that makes the worst-case overhead for uncontended access significantly higher. Indeed in any system that uses an optimistic version of PIP, and shared multiple resources (of any type) can use the proposed techniques to trade the number of resources versus schedulability. PIP will have favorable

schedulability properties compared to PCP due to the higher uncontended overhead.

The Stack Resource Policy (SRP) [18] enables the possibility of many tasks sharing a common run-time stacks. In contrast, we focus on splitting the execution contexts (stacks) for a thread across components for increased isolation, and managing the distributed stack resources.

Baruah [27] investigated a strategy for replicating shared resources to meet schedulability constraints in EDF scheduled systems using SRP. This work differs in the following ways. (1) We focus specifically on solving the tangible problem of how to manage run-time stacks in a system with pervasive fault boundaries, and base our results on measured parameters of the COMPOSITE component-based OS. (2) We compare multiple resource sharing protocols and identify system parameters that favor one protocol over the other. (3) Replication of resources partitions them between competing threads. In contrast, in our system, the amount of stacks available in a component can be changed, but they cannot be partitioned for specific threads. We provide analysis and allocation algorithms that account for this different problem formulation.

In this paper, we make the pessimistic assumption that all functions that can be invoked in a component's interface have the same worst-case execution time. Future work will explore more general models to differentiate a component's functions.

VIII. CONCLUSIONS

Motivated by systems decomposed into separate memory-isolated components, this paper presents the trade-offs between 1) system schedulability and stack memory usage, and 2) PCP and PIP for stack contention management in a real component-based system. Although PCP is typically formulated to be analytically superior to PIP, we find that real-world overheads for both policies (measured with an implementation in the COMPOSITE component-based OS) result in *more systems being schedulable under PIP than PCP when memory can be allocated for execution stacks.* Even when memory is scarce and each component has a single stack, PIP provides superior schedulability for some system configurations – notably those with more component dependencies, more tasks, or higher proportions of component execution to invocation overheads. We conclude that the choice of PIP vs. PCP for multi-unit resources (*i.e.* for managing execution stack contention) is complex and dependent on practical implementation overheads that must be considered in system design and analysis.

In contrast to traditional resource contention models in which blocking overheads are assumed as parameters to the model, we provide schedulability conditions for both PCP and PIP given a variable number of stacks allocated to each component. We define and evaluate algorithms for assigning these stacks attempting to minimize memory usage, while finding schedulable configurations. We find that in contrast to a naive stack allocation method, the system achieves schedulability while significantly cutting down the amount of memory necessary for stacks.

Source code for the system can be found on the COMPOSITE webpage at www.seas.gwu.edu/~gparmer/composite.html

Acknowledgements. We'd like to thank Andrew Sweeney for the initial implementations of the stack manager, and our shepherd, Daniel Mosse, along with the anonymous reviewers, for improving the quality of this paper.

REFERENCES

[1] J. Liedtke, "On micro-kernel construction," in *SOSP*, 1995.
[2] G. A. Parmer, "Composite: A component-based operating system for predictable and dependable computing," Ph.D. dissertation, Boston University, Boston, MA, USA, Aug 2009.
[3] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz, "The pebble component-based operating system," in *Proceedings of Usenix Annual Technical Conference*, 2002.
[4] I. Pyarali, M. Spivak, R. Cytron, and D. C. Schmidt, "Evaluating and optimizing thread pool strategies for real-time corba," in *LCTES*, 2001.
[5] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, 1990.
[6] M.-I. Chen, "Schedulability analysis of resource access control protocols in real-time systems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1991.
[7] U. Steinberg, A. Bottcher, and B. Kauer, "Timeslice donation in component-based systems," in *OSPERT*, 2010.
[8] U. Steinberg, J. Wolter, and H. Hartig, "Fast component interaction for real-time systems," in *ECRTS*, 2005.
[9] G. Parmer and R. West, "Predictable interrupt management and scheduling in the Composite component-based system," in *RTSS*, 2008.
[10] G. Parmer, "The case for thread migration: Predictable ipc in a customizable and reliable os," in *OSPERT*, 2010.
[11] G. Parmer and R. West, "Hires: A system for predictable hierarchical resource management," in *RTAS*, 2011.
[12] G. A. Parmer and R. West, "Mutable protection domains: Towards a component-based system for dependable and predictable computing," in *RTSS*, 2007.
[13] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in linux," in *Ottawa Linux Symposium*, 2002.
[14] J. H. Anderson, S. Ramamurthy, and K. Jeffay, "Real-time computing with lock-free shared objects," *ACM Trans. Comput. Syst.*, vol. 15, no. 2, pp. 134–165, 1997.
[15] B. N. Bershad, D. D. Redell, and J. R. Ellis, "Fast mutual exclusion for uniprocessors," in *ASPLOS*, 1992.
[16] G. Parmer and J. Song, "Customizable and predictable synchronization in a component-based os," in *the Conference on Embedded Systems and Applications (ESA)*, 2010.
[17] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo, "A new kernel approach for modular real-time systems development," in *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.
[18] T. P. Baker, "A stack-based resource allocation policy for realtime processes," in *RTSS*, 1990.
[19] H. Levy, "Capability-based computer systems," 1984.
[20] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, vol. 29, no. 5, May 1986.
[21] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, pp. 284–292, 1993.
[22] M. Sjödin and H. Hansson, "Improved response-time analysis calculations," in *RTSS*, 1998.
[23] G. Parmer and R. West, "Hijack: Taking control of cots systems for real-time user-level services," in *RTAS*, 2007.
[24] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann, "Interface and execution models in the fluke kernel," in *OSDI*, 1999.
[25] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy, "Lightweight remote procedure call," *ACM Trans. Comput. Syst.*, vol. 8, no. 1, pp. 37–55, 1990.
[26] "Scheduling in k42, whitepaper: http://www.research.ibm.com/k42/white-papers/scheduling.pdf."
[27] S. K. Baruah, "Resource sharing in edf-scheduled systems: A closer look," in *RTSS*, 2006.