Customizable and Predictable Synchronization in a Component-Based OS

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

Abstract

Component-based operating systems enable embedded systems to adapt system policies, mechanisms, and abstractions to the specific workloads and contexts of each system. The scope of an embedded system developer to customize the software of the system is often limited by the kernel abstractions. For example, synchronization and scheduling policies are often constrained to the static few provided by the kernel. As time-management is an essential aspect of many embedded systems, there is motivation to enable these systems to configure synchronization policies to their needs.

In this paper, we present a component-based implementation of system synchronization policies in the COMPOSITE *OS. This implementation provides fault-isolation between applications, synchronization mechanisms, and system schedulers while maintaining high levels of performance. Empirical evaluation demonstrates that the proposed primitives have performance comparable to a highly optimized, but uncustomizable futex mechanism in Linux.*

1 Introduction

Embedded and real-time systems are deployed in a large variety of environments, many of which require specialized policies for managing the resources of the system. The required policies differ across systems. For example, a system anticipating high utilization and dynamic workloads might prefer an earliest-deadline first (EDF) scheduling policy that has a utilization bound of 1, whereas a resource constrained system with a fixed task-set might use Rate Monotonic Scheduling (RMS) and a simple fixed priority scheduler [9]. Additionally, real-time system requirements go beyond scheduling policies. Famously, the mars pathfinder rover required a dynamic update to its software to activate the priority inheritance resource sharing protocol to complete its mission [6].

Motivated in part by the desire to provide a canonical base system that is configured for specific systems, component-based operating systems enable the system abstractions, mechanisms, and policies to be defined by replaceable components [1, 4]. A component is an encapsulated instance of some specific functionality that exports an interface through which that functionality can be used by other components. Specialized components – including those defining scheduling policies, physical memory mapping, and event notification – are chosen to satisfy the goals of the system and applications. This configurability is used to scale from very simple systems with low resource utilization to complex systems such as performance-based web servers.

In addition to increased configurability, component-based operating systems often enforce the encapsulation of user-level components using protection domains (via, e.g. hardware page-tables). This increases the reliability of the system as the scope of the adverse effects of an errant component are limited to that component.

Though progress has been made to define many system resource management policies as components, this paper investigates how synchronization and resource sharing policies can be implemented efficiently and predictably in a component-based OS. Integral to this is the definition of the appropriate interfaces to access synchronization primitives, and the functional implementation of the synchronization mechanisms themselves. As the scheduling policies of the system change, different synchronization policies are used. Consequentially, if scheduling policies are customizable [11], it is beneficial for synchronization to be comparably specialized. Even for a single scheduling policies (e.g. fixed priority), there are many possible resource sharing protocols that can be used to produce a predictable system. For instance, priority ceiling protocols (PCP) prevents unbounded priority inversion [13] and deadlock, but requires that one know the maximum priority of all threads accessing each lock. On the other hand, the priority inheritance protocol (PIP) doesn't prevent deadlock, but does not require a-priori knowledge of thread priorities. It is clear that appropriate synchronization policies should be chosen given the characteristics and requirements of each individual system.

More broadly, the customization of the system's synchronization protocols is useful for debugging – by using deadlock detection algorithms – and when customized in conjunction with the scheduler policy, can even be used to prevent bugs [7], or provide deterministic multi-core execution [10].

Figure 1 is an example of a simple component-based system configured with only three components, a client application component, a synchronization component providing a lock abstraction, and a scheduler component that can switch between multiple threads executing in the components. Invo-

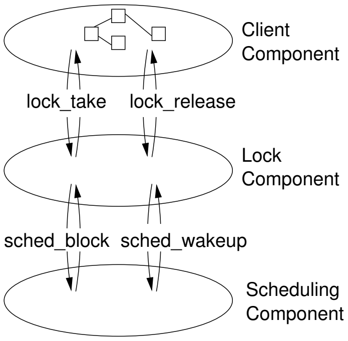

Figure 1. A client component invoking functions on a lock component which invokes scheduler functions.

cations between components are made on functions within the interface of the invoked component. If the components are in separate protection domains, this is mediated by the kernel, otherwise the invocation is made directly.

Challenges: The goals is to provide an efficient, customizable synchronization mechanism in a component-based system. As system schedulers change, or as the requirements for the system change, applications should be able to use specialized synchronization components. However, because components are in different protection domains, the cost of inter-component invocations must be considered to achieve acceptable performance. Additionally, many benefits of components rely on encapsulation of state, which can make consistency of state (e.g. lock state) across components difficult.

The *contributions* of this paper include

- the definition of different component-based synchronization policy implementations that consider both predictability and efficiency (esp. regarding component invocations), and
- an empirical evaluation of these protocols in the COMPOSITE component-based operating system, including an analysis for when each separate implementation is preferable.

Previous component-based systems typically define a kernel-provided (thus fixed) semaphore abstraction, or use threads to serialize access, which is limited by the default thread communication semantics [1, 8]. This paper, thus presents the first implementation we know of for configurable synchronization policies defined as possibly fault-isolated components. In this paper, we will focus mainly on the uniprocessor case as many embedded systems fall into this category. Additionally, this focus will simplify the discussion.

This paper is organized as follows: We introduce the COMPOSITE component-based OS in Section 2, and the methodologies for providing component-defined synchronization policies in Section 3. In Section 4, we experimentally evaluate the synchronization policies. In Section 5 we discuss the related work, and Section 6 concludes.

2 The COMPOSITE Component-Based OS

To investigate the design of a component-based implementation of synchronization policies, we use the COMPOSITE component-based operating system [12]. The system policies, mechanisms, and abstractions that are typically found in the kernel in traditional systems, are implemented as user-level components in COMPOSITE. For example, scheduling policies, networking protocols, physical memory management, and, of course, the synchronization policies, are defined as user-level components. Invocations of a specific function in a component's interface are used to access that component's functionality.

Threads and Schedulers: Threads begin execution in a component, and can invoke others throughout their lifetime. Invocations mimic function calls, thus threads execute through many components to harness the system's functionalities. System schedulers are normal user-level components that have the capability to dispatch (switch) between threads. This capability is used to block threads (switch away from them until some event occurs), and to wake them up when the event occurs (switch to them). Component-based schedulers afford a number of benefits. (1) System temporal policies are configurable and can be replaced by simply using a different component, and (2) schedulers are isolated from faults in other components, and likewise. This can increase the dependability of the system if appropriate recovery techniques are used [3, 2].

Protection Domains: Each component is, by default, isolated from other components in its own protection domain (provided by hardware page tables). Invocations between components require switching between page-tables, and are therefore mediated by the kernel. Though this process is optimized, direct function invocations is still significantly faster. A performance comparison is conducted in Section 4. COMPOSITE enables the system to manage the trade-off between fault-isolation (inter-protection domain invocations), and performance (direct invocation) with a mechanism called Mutable Protection Domains (MPD) [12]. MPD enables protection domain boundaries between specific components to dynamically be constructed and removed in response to where the overheads for inter-protection domain invocations are the greatest. Importantly, where security constraints exist in the system (e.g. between applications), protection boundaries always exist to ensure system security. Contrarily, if a security boundary doesn't exist between components, the main purpose of protection domains is to increase reliability by limiting the scope of errant behaviors.

2.1 Scheduler Interfaces

Component Critical Sections. A component that provides a lock abstraction will require critical sections itself to modify locks structures, do memory allocation, etc. Here we describe what facilities schedulers provide in COMPOSITE towards this. Schedulers provide a single lock per component that can be used to create critical sections for that component. Components (e.g. lock components) can use this to define their own higher-level synchronization primitives. Building higher-level synchronization primitives on those that are lower-level is common: monolithic kernels use the ability to disable interrupts to construct mutexes, and use these mutexes to provide user-level accessible synchronization primitives.

Specifically, Schedulers provide a pair of functions: `sched_comp_critsect_take(comp_id)` and `sched_comp_critsect_release(comp_id)` that a lock component can use. Each component can invoke these functions to provide mutual exclusion between the `take` and `release`. If a thread invokes the `sched_comp_critsect_take` function, and another thread is currently holding the critical section, the scheduler will immediately switch to the critical section holder allowing it to complete its critical section. Upon releasing the critical section, the scheduler immediately switches to the contending thread, thus allowing it to take the critical section. The goal of the scheduler here is to provide a simple mechanism such that the other components can define their own customizable, and semantically rich synchronization policies.

Controlling Thread Execution State. Schedulers provide two other functions that are used to control thread execution, `sched_block(thd_dep)` and `sched_wakeup(thd)`. `sched_block` blocks the current thread [1](i.e. the one calling `sched_block`), and creates a dependency between it and the thread identified by `thd_dep`. The scheduler deals with dependencies as follows: if a thread τ is dependent on τ^d, and the scheduling policy wishes to execute τ, it will immediately switch to τ^d instead. τ^d will presumably switch to τ at its earliest convenience which will remove the dependency (as is exemplified in Algorithm 1).

To wakeup a thread that has been blocked, the `sched_wakeup` function is used. The scheduler will make the thread runnable and remove any thread dependencies it may have. It will be executed the next time the scheduling policies wishes for it to run.

3 Synchronization Component Design and Implementation

This paper investigates how synchronization policies can be specialized for different subsystems and applications. The basic goals of any synchronization primitive's design include:

• **Predictability:** The basic implementation should not include unbounded operations such as memory allocation [2]. Additionally, we will discuss the ability to customize synchronization policies to include features necessary in embedded systems such as *preventing unbounded priority inversion*, and more generally, enabling the definition of customized policies.

• **Efficiency:** As synchronization primitives might be used quite frequently, the overhead they impose should be as small as possible. This might include optimising for different common cases (e.g. often critical sections are not contended by multiple threads).

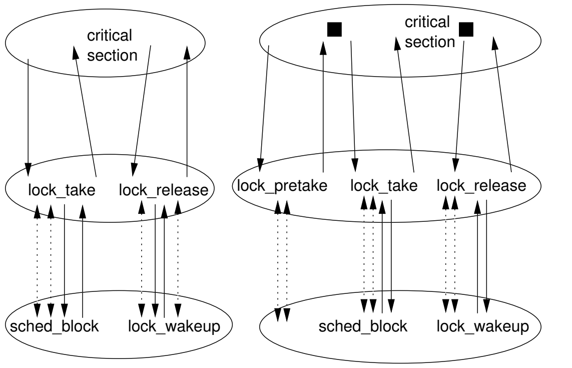

Figure 2. Invocations made between components for the pessimistic and optimistic algorithms assuming contention when taking and releasing the lock. *Left:* pessimistic lock component. *Right:* optimistic lock library and component. Dotted invocations designate invocations of `sched_comp_critsect_take` and `sched_comp_critsect_release`. The black square indicates atomic instructions.

We will discuss two lock implementations. Figure 2 depicts a pessimistic implementation, and an optimistic implementation that optimises for uncontended use of locking primitives.

3.1 A Pessimistic Lock Component

We now discuss the implementation of a synchronization component that provides a lock (or mutex) abstraction to client components. This component is *pessimistic* in that the implementation is not optimised for locks that are uncontended [3]. The uncontended case is not treated specially. Many operating systems have up until recently provided (or still provide) comparible user-level locking primitives. The equivalent lock in monolithic systems would require a system call for both the taking and releasing of each lock. In COMPOSITE, this means that each acquisition and release of a lock involves an invocation (IPC) between the client component, and the lock-providing component.

The pessimistic lock (or `plock`) implementation is detailed in Algorithm 1. These two functions provide the traditional operations of taking and releasing a lock. Additional functions for allocating and freeing a lock are not displayed. These trivially involve the allocation and deallocation of the lock data-structure within the synchronization component. Locks are identified in the client component as an integer (`lock_id` in the algorithms).

Each lock structure (denoted `local_lock`) includes a field for storing the owner of the lock. If a lock isn't taken, then the owner is set to `0`. In `lock_take`, if the lock is already taken by another thread, then we wish to (1) add the current thread to a list of the *blocked* threads for that lock, and (2) block the current thread with a dependency on the thread who owns the lock.

[1] The thread will block *unless* a wake-up for it has been received in the mean time. This might happen if a thread is preempted before completing an invocation to `sched_block`.

[2] It is difficult to bound the execution of the out-of-memory case.

[3] This should not be confused with optimistic or pessimistic concurrency mechanisms such as transactions versus mutexes.

Algorithm 1: Pessimistic Lock

```
lock_take:
Input: lock_id: integer identifying the specific lock
sched_comp_critsect_take(this_comp)
local_lock = get_lock_datastruct(lock_id)
owner = local_lock.owner
if owner == 0 then
    local_lock.owner = this_thd_id
    sched_comp_critsect_release(this_comp)
    return
blocked_list_node.thd = this_thd_id
add_list(local_lock.list, blocked_list_node)
sched_comp_critsect_release(this_comp)
sched_block(owner)
// now we are the lock owner
return

lock_release:
Input: lock_id: integer identifying the specific lock
sched_comp_critsect_take(this_comp)
local_lock = get_lock_datastruct(lock_id)
if empty(local_lock.list) then
    blocked_list_node = dequeue(local_lock.list)
    blocked_thd = blocked_list_node.thd
    local_lock.owner = blocked_thd
    sched_comp_critsect_release(this_comp)
    sched_wakeup(blocked_thd)
else
    local_lock.owner = 0
    sched_comp_critsect_release(this_comp)
```

Algorithm 2: Optimistic Lock (Client Library)

```
lock_take:
Input: lock: lock structure
repeat
    repeat
        l_old = lock
        owner = l_old.owner
        if owner then
            l_new.owner = owner
            l_new.contended = 1
            lock_component_pretake(lock.id)
        else
            owner = l_new.owner = this_thd_id
            l_new.contended = 0
    until cmp_and_swap(lock, l_old, l_new)
    if owner != this_thd_id then
        // wait till owner releases the lock
        lock_component_take(lock.id, owner)
until owner == this_thd_id
return

lock_release:
Input: lock: lock structure
repeat
    l_old = lock
    contended = l_old.contended
    l_new.owner = 0
    l_new.contended = 0
until cmp_and_swap(lock, l_old, l_new)
if contended then
    lock_component_release(lock.id)
```

When releasing a lock, the owner checks if there are any threads blocked waiting for the lock to be released. If there are, the first one is woken up which then owns the lock.

Discussion: The pessimistic strategy is functionally correct and simple, but it does not optimize for the common case: when a lock is taken and released without contention. Indeed, to increase the efficiency, the next strategy considers that case specifically.

3.2 Optimistic Lock Component

In this section we discuss the implementation of a lock that optimises for the case when there is no contention (i.e. it is optimistic that there will be infrequent contention). In our experience with constructing a webserver on COMPOSITE, this is a worthwhile optimization [12]. We are inspired by Linux futexes [5] [4] that are designed to significantly decrease uncontended lock access latency for user-level programs by avoiding system calls.

The algorithm for the optimistic lock (or `olock`) consists of two main bodies of code, one that is loaded as a small library into the client component (Algorithm 2), and one in the lock component that defines lock behavior under contention (Algorithm 3).

The library must use some synchronization mechanism. As it is optimistically attempting to avoid making component invocations, the client library relies on an atomic instruction such as *compare and swap* to atomically read and update a memory location. Here we assume that compare and swap takes three arguments: the address to be modified, the value the thread last loaded from that memory location, and the new value that location is to be updated to. Compare and swap checks that if the memory location is set to the first value, and if it is, it is updated to the new value. This is done atomically, and it will return true if the update is made, false otherwise. Instead of using architectural compare and swap instructions, we use restartable atomic instructions to provide the same functionality with less overhead as described in [11]. This technique can be used on architectures that do not provide atomic instructions. At a high-level, compare and swap is used to manipulate a word in memory semantically associated with the lock that includes both the owner of the lock, and a bit for if that lock has been contested or not (i.e. if there are threads blocked waiting for the lock to be released).

All manipulations for the common-case of uncontested critical section access are done within the library, avoiding component invocations. This enables the overhead of lock access in the uncontended case to approach that of a function call. However, it does complicate the contested case. This is because the relevant state concerning if a lock is currently taken or not, and if there are threads blocked on it, is distributed across multiple components (the client and the lock components). With distributed state, additional logic must ensure that actions in each component are taken with a consistent view of the lock's state.

Algorithm 3 depicts the code in the synchronization component for interfacing with the library in Algorithm 2. Much of the design of this algorithm is motivated by the need to enable a consistent view of the lock's state between the client and synchronization components. The main difficulty here

[4] Futexes are used by default if you use `pthread_mutexes`.

Algorithm 3: Optimistic Lock (Component)

```
lock_component_pretake:
Input: lock_id: integer identifying the specific lock
sched_comp_critsect_take(this_comp)
local_lock = get_lock_datastruct(lock_id)
local_lock.epoch = curr_epoch
sched_comp_critsect_release(this_comp)
return

lock_component_take:
Input: lock_id: integer identifying the specific lock
sched_comp_critsect_take(this_comp)
local_lock = get_lock_datastruct(lock_id)
if local_lock.epoch != curr_epoch then
    // outdated attempt to take the lock, has
       been released in the mean time
    sched_comp_critsect_release(this_comp)
    return
// block this thread
blocked_list_node.thd = this_thd_id
add_list(local_lock.list, blocked_list_node)
sched_comp_critsect_release(this_comp)
sched_block(owner)
// try to reacquire the lock in library again
return

lock_component_release:
Input: lock_id: integer identifying the specific lock
sched_comp_critsect_take(this_comp)
local_lock = get_lock_datastruct(lock_id)
curr_epoch ++
while !empty(local_lock.list) do
    blocked_list_node = dequeue(local_lock.list)
    blocked_thd = blocked_list_node.thd
    // release component lock before waking
       last thread
    if empty(local_lock.list) then
        sched_comp_critsect_release(this_comp)
    sched_wakeup(blocked_thd)
end
```

is typified by the following sequence: If a thread attempts to take a lock, but finds it contested, it will invoke the lock component to block waiting. However, if before the lock component can process this request to block, the lock owner releases the lock, then we are in a difficult situation. When resumed, the first thread will ask to block waiting for the lock to be released even though it is really available. The lock is *not* taken, so blocking it would be in error. The root problem is that the thread asking to block till the lock is released is making that request on ”stale” information (it believes the lock is taken, but it is actually not). There would be no way for the lock component to discriminate between this stale invocation and the typical condition where a thread is blocking on a lock that actually is current held by another thread. This condition, then, must be detected.

To disambiguate between ”stale” blocking requests from the client library, and requests to block on an actually contended lock, we identify the case where a lock has been released *after* a thread determines in the library code the lock is contended, but *before* it has successfully been placed on the list of blocked threads for the lock in the synchronization component. This is done by making multiple invocations of the synchronization component, one before the lock is set as contested, and one after. The synchronization component simply tracks if a release of the lock occurs between these invocations. It does so trivially by incrementing a value (an integer: `curr_epoch`) for each lock release. The epoch number is recorded before the lock is set as contended, and verified to be identical when the contending thread asks to block waiting for the lock to be released. If these values differ, then a release occurred between these calls. In this way, the lock component can detect if an invocation that is made to block the current thread is made on inconsistent information, i.e. when the lock isn’t actually taken.

There is an additional difference between this implementation and the one that doesn’t optimize for the uncontended case. The `lock_component_release` function – called when the lock is contended and released – wakes up *all* threads blocked waiting for the lock to be released. This in contrast to the pessimistic lock where only the first one on the list is unblocked. When they execute, they will again attempt to take the lock. This is an optimization assuming that the thread to next hold the lock will be able to release it before preempted. Subsequent accesses will be uncontended.

Discussion: The optimistic lock optimizes for the uncontended case, but pays in complexity, and in the cost of contended accesses. The invocation of `lock_component_pretake` to ensure consistency of lock state between client library and lock component increases the number of invocations. The optimal lock implementation, thus, depends on the workload.

3.3 Real-Time Resource Sharing Protocols

We have presented two algorithms for implementing locking in a component-based system. Here we describe how these are expanded to implement the priority inheritance protocol (PIP) and the priority ceiling protocols (PCP) to prevent unbounded priority inversion for real-time systems. We assume the reader is familiar with these protocols [13]. The point here is to demonstrate the presented implementations can be easily specialized to specific system requirements.

Priority Inheritance Protocol: Both existing algorithms automatically behave equivalent, semantically, to PIP. This is due to the dependencies that are tracked by the scheduler. PIP ensures that the lock holder is scheduled with the priority of the thread contending that lock with the highest priority. Thread dependencies define the following semantics: if a high-priority thread is chosen to run by the scheduling policy, the thread it is dependent on will actually be executed [5]. Thus, as long as when a thread blocks, it passes the holder/owner of the lock as a dependency, the behavior is equivalent to the PIP protocol.

Priority Ceiling Protocol: The pessimistic algorithm is trivially extended to PCP. Within `lock_take`, the scheduler must be invoked with a request to compare the current thread’s (possibly elevated) priority to the current ceil-

[5] The dependency relation is transitive, so a chain of threads with dependencies will determine a single thread the rest depend on. A cycle of dependencies denotes deadlock and is detected and treated as erroneous.

ing maintained by the lock component. Only if the lock is not taken, and the effective priority is higher than the ceiling will the lock be taken. In `lock_release`, the scheduler must again be invoked to lower the priority of the thread to its original level.

As the admission test to allow a thread to take a lock in PCP includes complex logic and invocations to the scheduler to retrieve thread priorities, it does not make sense to alter the optimistic lock implementation to accommodate PCP. If PCP is the desired protocol, the pessimistic lock better accommodates it.

4 Experimental Evaluation

We have implemented the proposed algorithms in the COMPOSITE component-based OS. To test the overhead of the approaches, we use a 1 Ghz Pentium M processor with 384MB RAM. For all experiments, the machine is quiescent during experiments. We compare against futexes in Linux version 2.6.20, and glibc version 2.7 with NPTL. Unless otherwise noted, we repeated measurements 1024 times for each experiment, reporting the average. We did not notice a significant change in the results with a higher number of iterations.

In the experiments, we investigate three components, a client component that relies on the synchronization component, that in turn relies on the scheduling (lock) component (as in Figure 1). These are denoted by c, l, and s. A specific protection domain configuration between these components is denoted by parenthesis. If two components are in the same set of parenthesis, Mutable Protection Domains is used to remove the intervening protection boundary, thus increasing performance. For example $(c)(l)(s)$ means that they are in separate protection domains, and $(c)(ls)$ denotes that the synchronization and scheduling components are co-located in the same protection domains.

It might seem dangerous to place both the locking and client components into the same protection domains as the scheduler. However, COMPOSITE supports hierarchical scheduling. This means that a *parent* scheduler can delegate scheduling duties of a subset of its threads to specific *children* schedulers. Thus it is possible for every application to have its own scheduler. In such a case, no security boundary is violated by placing that scheduler in the same protection domain as the rest of the application components.

For context, an inter-protection domain invocation in COMPOSITE takes 0.685 μ-seconds, and an invocation between components in the protection domain takes 0.009 μ-seconds. By comparison, an RPC between processes via a pipe in Linux takes 4.446 μ-seconds. Each value is the average of 100,000 measurements. A direct comparison to Linux should not be made as Linux is not optimised for IPC. Instead, these values should be compared only for context.

To determine the costs of the discussed synchronization mechanisms in COMPOSITE, we study the costs of taking and releasing a lock around an empty critical section. In Figure 3, the costs of uncontended access are measured for pessimistic locks (`plocks`), optimistic locks (`olocks`), and futexes. For many workloads, this is the common case where locks are used to ensure mutual exclusion, but the lock is taken and released before another thread can contend the lock. Each value represents the average of 10,000 lock/unlock pairs. For both the optimistic locks and futexes, the configuration of protection domains does not effect performance as no invocations are made in the uncontended case. Though the pessimistic implementation does require invocations of the lock component, it does not require any thread switches. The scheduler component is still invoked to provide critical sections for the lock component. The optimistic implementation is faster than futexes as it requires only a single atomic operation whereas futexes must synchronize with the kernel and are more complicated [5].

Figure 4 plots the latency of the various implementations when there is contention for the lock. A high priority thread attempts to take a lock held by another thread. This thread is immediately switched and it releases the lock, letting the higher priority thread take it. The latency of this operation is measured.

When the components are in separate protection domains, the optimistic locking implementation has a higher overhead than the pessimistic one. This is due to the invocation overheads for `lock_component_pretake`, and its invocations of the scheduler. Both the optimistic lock, and futexes require more logic to ensure that the lock really is contested than the pessimistic case, thus why the (cls) version of the pessimistic lock is fastest.

The pessimistic lock is more efficient than the optimistic lock under contention, and the optimistic lock is significantly more efficient when there is low contention. This again motivates the component-based approach which provides the ability to customize your locking implementation to the specific workload of your system and applications. Figure 5 plots the average latency for lock operations given a variable proportion of uncontended to contended accesses. The x axis denotes this proportion (i.e. $\#\, contended\, accesses/total\, \#\, accesses$). Because of the low uncontended cost, the optimistic lock implementation is the more efficient than the pessimistic one up to when 33.5%, 57.7%, and 62.6% of the accesses are uncontended for (cls), $(c)(ls)$, and $(c)(l)(s)$, respectively. When there are higher contention rates, the pessimistic implementation is more efficient.

When compared to a system without customizable locking policies (Linux with futexes), the optimistic case is faster when 12.5% or less of accesses are uncontended (for (cls)). When there is more than 43% contention, the pessimistic lock implementation has less overhead than futexes (again for (cls)). The emphasis of this work is to provide customizable synchronization policies with acceptable performance. As the efficiency is generally in the same magnitude as commonly used mechanisms, we consider this a success.

5 Related and Future Work

Many component-based or μ-kernel systems [8, 1, 4] have attempted to increase the configurabiliy and reliability of the system by structuring the system around components or

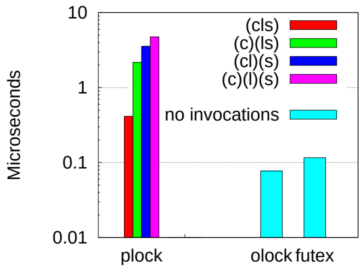

Figure 3. Uncontended lock latency (logscale).

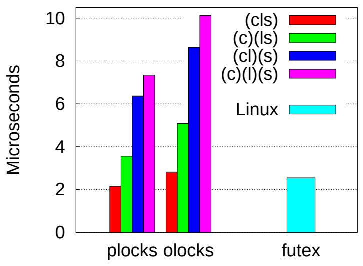

Figure 4. Contended lock latency.

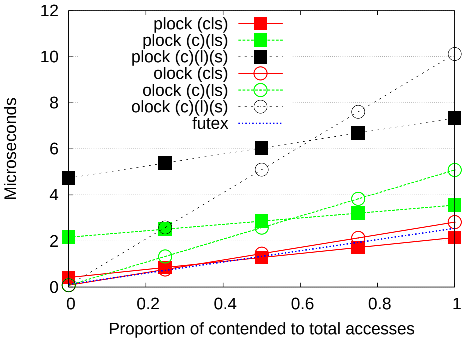

Figure 5. Lock latency for proportion of contended lock access to total

servers. COMPOSITE has increased the degree to which this is possible by removing scheduling policy from the kernel and defining it in components [11]. Whereas previous systems have used a kernel-provided semaphore abstraction, or thread serialization which are both inflexible and limited by kernel policies, we seek to define all synchronization via user-level components. This increases the scope of system customization by enabling specialization of synchronization mechanisms.

The implementation strategies we investigate are inspired by many previous approaches. The pessimistic approach to locking is similar to distributed algorithms for mutual exclusion whereby a centralized server accepts requests for clients to enter the critical section, and replies in turn to those that can access it. We observe that a single client component shares memory between different synchronizing threads, and this leads to the optimization of the uncontended case where all modifications of the lock state are made in the client. This design was in some ways inspired by futexes [5]. The implementation differs from futexes in that we do not assume that a region of memory is shared between the client and the locking component [6] as such sharing would weaken the reliability constraints of the system.

In the future, we will consider further improving the performance of these locks. One promising approach to use an optimistic locking component to provide the necessary critical sections for *a higher-level locking component* instead of using the scheduler critical section. This will remove, in the common case, most invocations from the lock component to the scheduler and will significantly improve the access latency for the optimistic lock under contention.

6 Conclusions

We detail the design and implementation of component-based synchronization protocols in the COMPOSITE component-based operating system. We show that two different implementations can optimize for different usage cases, and describe how they are expanded to include typical resource sharing protocols. This enables the customization of system synchronization policies for the requirements of the environment or the application. We empirically evaluate these implementations and show that their overhead is on par with more traditional mechanisms that do not support customization.

The COMPOSITE source code is available upon request.

References

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

[2] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot–a technique for cheap recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–44, December 2004.

[3] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. Curios: Improving reliability through operating system structure. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, CA, December 2008.

[4] J. Fassino, J. Stefani, J. Lawall, and G. Muller. Think: A software framework for component-based operating system kernels. In *Proceedings of Usenix Annual Technical Conference*, June 2002.

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

[6] M. B. Jones. What really happened on mars?, 1997.

[7] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock Immunity: Enabling Systems To Defend Against Deadlocks. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

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

[9] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 1973.

[10] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 97–108, 2009.

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

[12] G. A. Parmer. *Composite: A Component-Based Operating System for Predictable and Dependable Computing*. PhD thesis, Boston University, Boston, MA, USA, Aug 2009.

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

[6] Futexes are mapped both in the kernel and user-level.