The Case for Thread Migration: Predictable IPC in a Customizable and Reliable OS

Gabriel Parmer

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

Abstract

Synchronous inter-process communication (IPC) between threads is a popular mechanism for coordination in μ-kernels and component-based operating systems. Significant focus has been placed on its optimization, and consequently the efficiency of practical implementations approaches the lower limits set by hardware. This paper qualitatively compares the predictability *properties of the synchronous IPC model with those of the migrating thread model. We assess the idealized communication models, and their practical implementations both in different versions of L4, and in the* COMPOSITE *component-based OS. We study three main factors – execution accounting, communication end-points, and system customizability – and discuss the trade-offs involved in each model. We make the case that the migrating thread model is as suitable as synchronous IPC, if not more so, in configurable systems requiring strict predictability.*

1 Introduction

Component-based operating systems are an appealing foundation for embedded and real-time systems as they enable high degrees of system specialization and enhanced reliability. The system's software is decomposed into fine-grained components. Each component provides a policy, abstraction, or mechanism that is accessed by other components through its interface. A system with specific timing constraints, or that is reliant on specific resource management policies, chooses the appropriate components to satisfy those particular requirements. By segregating components into separate protection domains (provided by e.g. hardware page-tables), the reliability of the system is increased as the scope of the side-effects of faults is limited to individual components. Communication and interaction between components is conducted via inter-process communication (IPC) in which the kernel mediates control transfer between protection domains.

Many different IPC mechanisms exist including synchronous IPC between threads, and thread migration. Implementations using synchronous IPC[1] exist that are extremely efficient, approaching the performance lower-bound imposed by hardware [10]. This method is used in many systems focusing on extreme IPC performance [16, 20, 17, 19], and it is employed in at least two commercially successful OSes, QNX and OKL4[2]. To accomplish most tasks, coordination between system components is required. Thus the predictability of the IPC operation impacts the real-time characteristics of all software in a component-based system. This paper seeks to provide a qualitative analysis and comparison of the predictability properties of this established IPC mechanism with those of thread migration [7]. For invocation using thread migration, a single schedulable thread executes across components.

We base most comparisons in this paper on, first, a pure model of synchronous IPC presented in Section 2.1, and, second, a variety of implementations of L4, a mature and highly-efficient μ-kernel [11]. We choose L4 as various implementations and optimizations have been made that demonstrate many interesting trade-offs in the design of synchronous IPC. As a concrete implementation of thread migration, we compare against the COMPOSITE component-based OS.

Functionally, both synchronous IPC and thread migration often look identical to client component code. Both are made to mimic normal function invocation by a interface

[1]A note on terminology: In this paper, we will refer to synchronous IPC between threads as simply *synchronous IPC*, and will use *IPC* and *invocation* interchangeably to denote control transfer back and forth between components. Additionally, we will use common terms in the μ-kernel literature to denote components as the *client* (making an invocation), and *server* (receiving and handling the invocation).

[2]See www.qnx.com and www.ok-labs.com.

definition language [4], hiding the concrete mechanisms used for the invocation. Behaviorally, they differ greatly and in this paper we focus on these differences. Many predictable systems have been built using synchronous IPC, but we argue here that thread migration is just as strong a foundation, if not more-so for predictable, configurable, and reliable systems. We base this argument on three main factors: (1) how processing time is accounted to execution throughout the system, (2) the effects of contention on the communication end-points in the system, and (3) the effect of the invocation mechanism on the ability of the system to provide configurable and specialized services.

This paper makes the following contributions:

• identify key factors that effect the predictability and flexibility of synchronous IPC;

• analyze the migrating thread model with respect to these factors, and compare against synchronous IPC;

• suggest a number of changes to a system built on synchronous IPC, that are inspired by the migrating thread model, to increase system predictability.

This paper is organized as follows: Section 2 introduces models to describe synchronous IPC and thread migration, so that they can be compared qualitatively. Section 3 discusses the different CPU allocation and accounting properties of both models, while Section 4 investigates the properties of IPC end-point contention, and Section 5 discusses system specialization and configuration opportunities present in the migrating thread model. Section 6 discusses the limitations of the migrating thread model, while Section 7 outlines related work, and Section 8 concludes.

2 IPC Models

2.1 Synchronous IPC Between Threads

Here we introduce an idealized version of the synchronous IPC model. Though many real-world implementations do not implement it directly, it serves as the starting point for their mechanisms. In the following sections we will discuss how various implementations diverge from this strict model where appropriate.

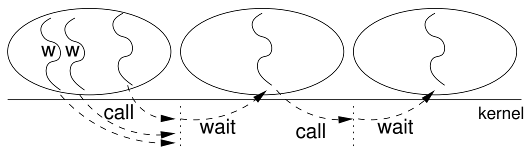

Figure 1. Synchronous IPC between threads. Threads annotated with a w are on a wait-queue associated with the server thread.

A system consists of a number of components, $C_a, \ldots, C_z$, each in separate protection domains. Thus communication between components must be conducted via the kernel (as switching between protection domains is typically a privileged instruction). Each component contains a number of threads $\tau_0^{C_a}, \ldots, \tau_n^{C_a}$. When thread $\tau_0^{C_a}$ wishes to harness the functionality provided by C_1, $\tau_0^{C_a}$ sends a message to $\tau_0^{C_b}$, and waits to receive a reply. This *send and receive* is often conflated into a single *call* system call. $\tau_0^{C_b}$ waits for requests from client threads, processes a request when one arrives, and replies to the client. The operations of *reply and wait* are often conflated into a single *reply_wait* system call. These API additions optimize for synchronous IPC and reduce the number of required user/kernel transitions [10]. If $\tau_0^{C_b}$ is processing and not waiting for an IPC when $\tau_0^{C_a}$ *call*s it, $\tau_0^{C_a}$ will block in a queue for $\tau_0^{C_b}$ [3], which will refer to as the *wait-queue* for a server thread.

Figure 1 illustrates synchronous IPC between three protection domains. A server thread, $\tau_0^{C_b}$, will become a client by harnessing the functionality of a third component and *call*ing $\tau_0^{C_c}$. We will say these nested IPCs create a *chain* of invocations. More generally, when taken together with the wait-queues for each server thread, a *dependency graph* [20] is created where threads waiting for a reply from a server thread (either because the server thread is processing on their behalf, or because they are in the wait-queue) are said to have a dependency on the server thread.

2.2 Thread Migration

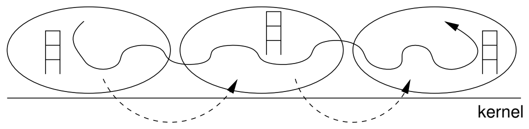

Figure 2. Thread migration. Execution contexts are spread across components, but the same schedulable entity traces invocations.

Thread migration [7] is a natural model for making invocations between components in a system. The same schedulable entity in one component continues execution in the other. The same thread, τ_0, executes through system components just as a thread in an object oriented language traverse many objects. If components are resident in the same protection domain this enables direct function invocation with little overhead [14, 15]. If system components exist in separate protection domains, then thread migration is less natural, but can still be accomplished. We will assume components in separate protection domains from now

[3]Note this is not the only option. The *call* system call can return an error code indicating the server thread is not ready for invocations. The consensus for synchronous IPC amongst surveyed implementations instead chooses the previous option.

on. In such a case, the *execution context* for a thread and the *scheduling context* are decoupled [20]. An invocation into each protected component requires a separate execution context (including C stack and register contents), but the scheduler treats the thread as a single schedulable entity. Figure 2 depicts thread migration.

2.2.1 Thread Migration in COMPOSITE

COMPOSITE is a component-based operating system focusing on enabling the efficient and predictable implementation of resource management policies, mechanisms, and abstractions as specialized user-level components [14]. Higher-level abstractions such as networking and file-systems are implemented as components, as are less-conventional low-level policies for task and interrupt scheduling [13], mutual exclusion mechanisms, and physical memory management. Components, by default, are spatially isolated from each other in separate protection domains (provided by hardware page-tables).

Components export an interface through which their functionality can be harnessed by other components. As system policies and abstractions are defined in components, invocations between components are frequent and must be both efficient and predictable.

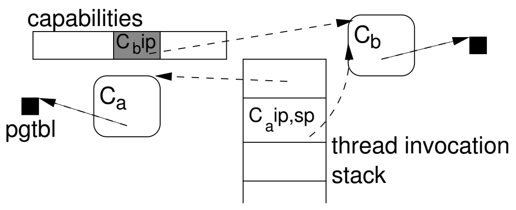

Figure 3. COMPOSITE kernel data-structures involved in an invocation. A syscall specifies a capability (associated with C_a) that yields the component to invoke (C_b). A thread's *invocation stack* saves the invoking component and enough state to return from the invocation (namely, the instruction and stack pointers).

The main kernel data-structures involved in an invocation between component C_a and C_b are depicted in Figure 3. Each component is restricted to make invocations only to components to which it has a capability [9]. Kernel capability structures link components and designate the authority to make invocations from one to the other. A thread executing in C_a that makes an invocation on a capability (via system call), will resume user-level execution in C_b, the component designated by the capability. This invocation occurs within the same thread, thus the same schedulable entity.

In addition to designating which component to execute in, a capability includes the instruction pointer in C_b to begin execution at. To maintain isolation, execution in C_b must be on a different stack from the one used in C_a. This execution stack in C_b is not chosen by the kernel. Instead, it is assumed that when the upcall is made into C_b, the first operation performed is to locate a stack to execute on. This operation we will refer to as *execution stack retrieval*. A simple implementation of this is to have a freelist of stacks in C_b, and to remove and use one upon upcall. Execution stack retrieval must be atomic to maintain freelist integrity as thread preemptions can occur at any time. COMPOSITE supports restartable atomic sequences [13] to provide this atomicity, even on processors that don't support atomic instructions. If the freelist of stacks is empty, then C_b invokes the *stack manager* component that either allocates a stack in C_b, or blocks the requesting thread until one becomes available.

As depicted in Figure 3, the structure representing a thread in COMPOSITE includes an *invocation stack*[4] which traces all invocations that have been made while in the context of that thread. Each entry in the stack includes a reference to the component being invoked, and the instruction and stack pointers to return to in the previous component. When an invoked component returns from an invocation (by invoking a static *return capability*), an item is popped off of the invocation stack, and the appropriate protection domain, stack pointer, and instruction pointer are loaded, returning execution to the invoking component (C_a). This process avoids loops, and doesn't touch user-memory so it shouldn't fault. The kernel invocation path, then, should be predictable.

Invocation arguments are passed in registers – up to 4 words on the x86 COMPOSITE implementation. Additional arguments are passed via shared memory.

IPC Efficiency in COMPOSITE: As in L4, the number of data-structures (thus cache-lines and TLB entries) touched during an invocation is small to minimize cache interference, and improve performance [10]. COMPOSITE's invocation path is implemented in C. It achieves performance on the order of optimized synchronous IPC paths also implemented in C. A component invocation takes less than 0.7 μ-seconds on both a 2.4 Ghz Pentium 4 processor, and a 1 Ghz Pentium M processor[5]. This is comparable to reported performance numbers in the μ-kernel literature [23, 15].

We believe that this demonstrates that thread-migration can be implemented without significant performance overheads compared to other techniques. Given this, the question is what are the other system factors that favor either

[4]Please note that this invocation stack is unrelated to the C stack.

[5]The average invocation overheads on these processors are similar – though they have varying clock speeds – due to significant differences in hardware overheads for user-kernel transitions, page-table switches, and relative CPU/memory speeds.

thread migration, or synchronous IPC. We investigate these in the rest of the paper.

3 CPU Allocation and Accounting

In this section, we investigate how CPU time is allocated amongst and accounted to different threads.

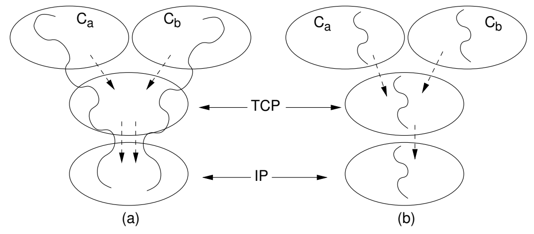

Figure 4. Invocations through components: (a) thread migration, (b) synchronous IPC.

We start with a simple system depicted in Figure 4. Application execution starts in a client component C_a and makes an IPC to C_{TCP} which, in turn, makes an IPC to C_{IP}. This could correspond to an untrusted client sending a packet through the transport and internetworking layers. Additionally, a second client component, C_b, causes the same progression of IPCs. Assume that C_a and C_b do not trust each other, and that they simply use the services provided by lower-level components.

Here we wish to investigate how CPU time is allocated and accounted throughout the system, and how it effects the policies for managing time. We investigate two models: synchronous IPC with a separate thread per component, and migrating threads where threads start in C_a and C_b and execute throughout system components. Figure 4 depicts these two situations.

From a resource allocation and accounting perspective, these two models are very different. To illustrate, assume that the amount of cycles spent processing in C_a is p_a^a. Invocations from this component result in p_{TCP}^a and p_{IP}^a cycles sent processing in C_{TCP} and C_{IP}, respectively. Additionally, the amount of time spent processing for the execution originating in C_b is p_b^b, p_{TCP}^b and p_{IP}^b, correspondingly.

3.1 Synchronous IPC Accounting and Execution

Client execution accountability: In the synchronous IPC model, the processing time spent in each component is charged to the component's thread. Thus the thread in the initial applications will be charged for their execution: p_a^a and p_b^b. However, the execution charged to $\tau_0^{C_{TCP}}$ will be $p_{TCP}^a + p_{TCP}^b$, and $\tau_0^{C_{IP}}$ will be charged for $p_{IP}^a + p_{IP}^b$. If the number of requests originating from C_a is significantly larger than those from C_b (even if the processing time in those components is small), the system scheduler will have little ability to throttle one client, or to even know which client is causing the overhead in the networking stack. The fundamental problem with this model for tracking CPU usage, and scheduling computation, is that it loses information about which client a shared component is doing processing for. Practical approaches to many of the shortcomings of the pure synchronous IPC model are discussed in Section 3.3.

Real-time task models: Aside from the inability of the scheduler to properly track client execution throughout the system, synchronous IPC does not naturally accommodate traditional real-time task execution models. It is common to assume a task has a given worst-case execution time, C, and executes periodically, with a period of T. C includes all execution time, including that which occurs in server components. The scheduler, will not see the thread using C execution time as the accounting for this execution is distributed throughout invoked threads in the system. This would make it difficult if not impossible to implement accurate aperiodic servers [22] that make invocations to other components, as budget consumption would be spread across multiple threads. An additional problem arises as the priority of a thread is often associated with its C (e.g. in rate-monotonic scheduling). It is not obvious how to assign priorities to threads throughout the system in the presence of pervasive thread dependencies. This is especially true in an open real-time system where an unknown number of nonreal-time or soft real-time tasks execute along side hard real-time tasks and they can all rely on shared servers. This problem only becomes more pronounced as the depth of the component hierarchy increases.

An application can avoid these problems by making no invocations to server threads. Unfortunately, this limits the functionality available to that application, and prevents the decomposition of the system into fine-grained components.

Priority Inversion: A server thread might have a low priority compared to a high-priority client. In such a case, a medium priority thread can cause unbounded priority inversion. To avoid these situations, great care must be taken in assigning thread priorities throughout the system. For example, [5] proposes a static structuring such that server threads always have the same or higher priority than their clients. Unfortunately, it is not clear if it generalizes in open systems. Additionally, as it requires that servers run at a higher priority, it can lead to larger scheduling interference of high priority server threads (that service predominantly low priority threads) with medium priority threads elsewhere in the system.

One might be tempted to observe that many of these problems come from having components that are relied upon and invoked by multiple other components, possibly with widely varying temporal requirements. Can't we simply arrange the system such that there are no components that are shared between different subsystems? Unfortunately, it is difficult to not share components that drive shared peripherals (e.g. keyboards, networking cards), that share the system's physical memory between subsystems, or that schedule system's threads (assuming component-based scheduling [13]). Such sharing is unavoidable.

3.2 Migrating Threads Accounting and Execution

The migrating thread model makes it explicit which client a server component is processing for, e.g. computation in the networking stack is performed in the scheduling context of the client thread. The scheduler explicitly sees all execution performed on behalf of a specific client, and can schedule it accordingly. Thus, the execution time accounted to the thread created in C_a is $p^a_a + p^a_{TCP} + p^a_{IP}$, and likewise for the thread created in C_b. If τ_a makes a disproportionately large amount of invocations into the networking stack, it is charged directly for the processing time of those invocations (in contrast to the synchronous IPC case).

Priority Inversion: A significant complication with the migrating thread model concerns shared resources *within* a server component. If a low-priority thread takes a shared resource requiring mutual exclusivity (e.g. it is protected by a lock) priority inversion can occur if it is preempted by a high-priority thread that attempts to access the shared server resource. The solution to this is to use a resource sharing protocol that bounds the priority inversion [18]. In COMPOSITE, locking policies including those that avoid unbounded priority inversion are implemented as components.

We claim that the resource management and accounting properties of this model more closely match the intended structure of a system composed of many components. There is some precedent for this position: When a user-level process makes a system call, the execution time spent in the kernel is typically accounted to and scheduled with the credentials of the user-level thread. That is to say, that threads migrate from user- to kernel-level (though, of course, their execution contexts change).

3.3 Synchronous IPC Implementations: Accounting and Execution

Actual implementations of synchronous IPC deviate from the pure model. In this section, we discuss the relevant differences.

In synchronous IPC, the kernel switches between threads on each IPC. It is thus natural to perform scheduling on every IPC. However, the overhead of scheduling decreases IPC performance significantly. An optimization is to use *lazy scheduling* to avoid scheduling until the scheduler is explicitly invoked (e.g. via a timer-interrupt), and to do *direct process switch* whereby the system switches directly to the server thread upon IPC [10, 17] (assuming the server thread was blocked waiting for an IPC) [6]. The combination of these techniques removes scheduling related overheads from the IPC path.

Unfortunately, The thread that is charged for execution at any point in time is not predictable. Before the execution of the scheduler, the invoking thread is charged, emulating migrating threads. However, after the scheduler is executed, the threads are scheduled separately. This unpredictability is harmful to real-time systems [16], and researchers have tested if the optimization is indeed necessary for efficiency [6]. The answer appears dependent on the frequency of IPCs. It should be noted that in such a case we are choosing between two undesirable cases: (1) unpredictable resource accounting and scheduling (via direct process switching and lazy scheduling), and (2) the problems associated with the pure synchronous IPC between threads (Section 3.1) including the associated overhead.

Side-stepping these problems, Credo [20] decouples the execution context and scheduling context of threads. Synchronous IPC between threads transfers the scheduling context to the receiving thread. This model can require walking a path in the *dependency graph* of thread synchronizations to maintain proper scheduling context assignments. Credo essentially moves the synchronous IPC regime towards a migrating thread model. Unfortunately, it does so at the cost of complexity, and it increases the worst-case execution time of invocations by requiring the walking of the dependency graph to determine current scheduling context. If the depth of this tree is not predictable, then IPC operations themselves will, in turn, not be predictable.

Discussion: Motivated by efficiency or better accounting, practical synchronous IPC implementations have moved towards the accounting and execution style of a migrating thread model. However, they do so a the cost of complexity and possible unpredictability. Systems requiring predictable IPC in which dependency graph depths cannot be statically known, would benefit from starting with a migrating thread model.

4 Communication End-Point Contention

IPC in μ-kernels and component-based OSes is directed at specific communication end-points. The end-point in synchronous IPC systems is the server thread. This thread is addressed directly from the client (i.e. by thread id), or indirectly via a capabilities [9]; the end-point is the same. For

[6]It should be noted that K42 provides synchronous IPC with direct process switch between *dispatchers* that are similar in many ways to system threads.

thread migration, the target of an invocation is the component, or protection domain, being invoked. This component can be addressed either by id, or indirectly by capability. The end-point of an invocation is important as it effects system behavior when there is contention (multiple concurrent invocations) to that end-point.

4.1 Synchronous IPC End-Point Contention

Unpredictable IPC overheads due to end-point contention: If multiple threads attempt to conduct synchronous IPC with an active server thread, they are placed in its wait-queue. When the server thread replies, the system executes the thread being replied to, or one of the threads on the wait-queue, depending on which of all of the threads has the highest priority. The execution cost of finding the next thread to execute, then, is linear in the size of the wait-queue. Thus to enable predictable IPC, the number of threads concurrently *call*ing a specific server thread must be bounded. The assumption is often that the duration of an IPC is short, thus the server thread will be preempted with only a small probability. Thus, the wait-queues should rarely grow to significant length. In general component-based systems in which even applications are decomposed into separate components, the probability of preemption in an invoked component is high, thus the consideration of wait-queue length is important[7]. Importantly, worst-case IPC costs must be considered in hard real-time systems.

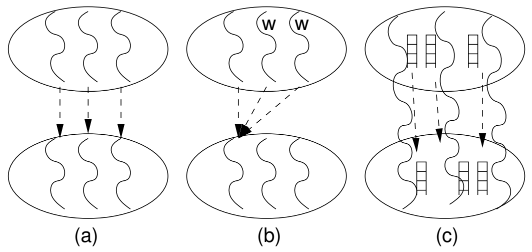

Figure 5. Invocations of and contention on various end-points. (a) All client threads invoke separate server threads. (b) Client threads invoke the same server thread, adding two to the server thread's wait-queue. (c) Thread migration: execution contexts aren't the target of invocation, thus cases similar to (b) are impossible.

Assume N threads concurrently attempt to invoke M threads in a server component. If $N = 1, M \geq 1$, then it is clear the IPC will continue without complication as the wait-queue is empty. If $M \geq N \geq 1$, and each of the N synchronizes with a separate server thread, the situation is comparable (Figure 5(a)). However, it is possible that all N invoking threads will attempt to synchronize with a single thread in the server, thus the wait-queue will be $N - 1$ long [8]. $M - 1$ server threads will remain waiting for IPC, and IPC overheads will correspondingly increase. This situation is depicted in Figure 5(b). If $N > M$, then some server thread's wait-queues will be unavoidably non-empty.

It follows that IPC predictability is dependent on if the following factors can be predicted: (1) the relative number of clients and server threads, and (2) the distribution of client invocations across server threads.

Limiting wait-queue length: Perhaps the most straightforward way to predict the maximum size of server wait-queues is to ensure that for each client thread, there is a corresponding server thread. Care is taken to only invoke a client's corresponding server thread. Though appealing in simplicity, this solution doesn't generalize for two reasons. First, the maximum number of threads in a protection domain is often bounded. Thus two components with the maximum number of threads each, would have at least twice the number of threads than are available in the server. Second, threads take up resources (e.g. memory). In the worst case such a strategy would require $T \times C$ threads, for T application threads and C components.

In a more realistic scenario, server threads are partitioned amongst different classes of client threads (with different priorities, or timing constraints). Fundamentally, client threads don't know the status of specific server threads (i.e. if specific server threads are busy or waiting for IPC). Yet on each invocation, they must answer "which server thread should I call?" Thus it is difficult, in the general case, for them to avoid invoking the same server thread.

Discussion: Predictable systems can be created using the two suggested modifications to synchronous IPC. However, in general systems with possibly malicious clients, and deep component hierarchies, it is not clear what the price of such techniques is (e.g. in memory consumption for thread context, or programmer complexity). Generally, the root problem is that the clients are forced to choose the specific execution context to process on in the server, but they don't have all information required to make that decision. That decision is best made by the server that knows its own state. In Section 4.3, we discuss possible enhancements to make this possible.

[7]Some synchronous IPC implementations disregard priority, and either switch immediately to the thread being replied to, or to the head of the wait-queue. This alleviates the problem of linear execution time in the size of the wait-queue. However, as it ignores thread priorities, it is unpredictable none-the-less.

[8]We are assuming a very specific interleaving of client threads where invocations are made before a server thread completes processing of an IPC request. This is the worst-case, and must be considered in real-time systems.

4.2 Thread Migration and End-Point Contention

For thread migration, the communication end-point being invoked is the server component[9]. As discussed in Section 2.2.1, when a component is upcalled into as the result of an invocation, the first operation it performs it to retrieve an execution stack from its local freelist. Assume N threads invoke the functions of a component in which M execution stacks (contexts) exist.

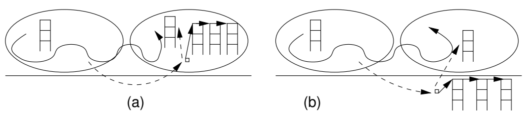

Figure 6. Retrieving execution contexts with thread migration. (a) Stacks are maintained on a freelist in the invoked component, or (b) in the kernel.

If $N \leq M$, all invocations will immediately find an execution stack to execute on. As the invocation end-point is the component, rather than specific execution contexts, so the server prevents contention on its stacks. As the execution contexts are not the target of IPC, the component has the opportunity to multiplex execution contexts as it deems appropriate. Clients will never block waiting for an execution context. Figure 6(a) depicts this operation.

If $N > M$, then contention for execution contexts is unavoidable, so the freelist of stacks will be empty for some invocations. In such cases, the thread invokes a component specializing in stack management. The stack manager allocates a new stack for immediate use, or calls the scheduler to block the thread until one becomes available. In the latter case, the stack manager implements priority inheritance to avoid unbounded priority inversion. Differentiated service between different clients or threads can be provided both at the time of execution stack retrieval, and in the stack manager by maintaining different lists of stacks for each level of service, and class of specific threads (i.e. hard real-time vs. best-effort execution contexts).

In COMPOSITE we choose to implement all policies for obtaining execution contexts at user-level in components. This enables (1) the definition of specialized policies as user-level components, and (2) the simplification of the kernel invocation path enabling its predictable execution and low number of data-structure accesses. It is possible to maintain the freelist of stacks in the kernel and assign a thread to a specific context for the duration of a component invocation. This is the approach taken by [8] which slightly complicates both kernel data-structures and the invocation path. Additionally, it places the policy for managing the stack freelist in the kernel (thus precluding the differentiated service policy described above). The benefit of this approach is that, in the case there is no available stack, it avoided the invocation into the server component. Figure 6(b) depicts this scenario.

[9]More specifically, in COMPOSITE the end-point is a function within the API of the component denoted by a capability is the target.

Discussion: By changing the target of invocations from individual threads in the server to the server itself, thread migration enables the server to manage and allocate its own execution contexts. This avoids multiple client threads waiting on a single server thread while other server threads are available which will increase IPC overheads.

4.3 Synchronous IPC End-Point Enhancements

Here we propose methods for modifying synchronous IPC implementations to include many of the benefits of the migrating thread model by changing the server communication end-point.

Locating execution contexts: One benefit of the migrating thread model is that the IPC end-point is not a specific execution context, thus the system – or the invoked component – has the opportunity to choose the appropriate context itself according to specialized policies. The system (and application) designer need not carefully plan which specific client and server threads communicate with each other. We believe that slight modifications to synchronous IPC implementations would enable the same capability.

Some modern μ-kernel systems [9] use capabilities to indirectly address the thread endpoint for IPC. Given this level of indirection, it would be natural for the capability to reference not a single thread, but a collection of server threads. Figure 6(b) depicts a similar scheme. Whenever an invocation is made with the capability, a thread is dequeued and execution in the server is made on that thread. As capabilities currently hold a pointer to the thread to IPC to, the overhead of this approach should be minimal. Capabilities can include a wait-queue of threads waiting to complete IPC with one of the server threads. Alternatively, when no server thread is available to service a *call*, a exception IPC (similar to page-fault IPC) can be delivered to a corresponding execution context manager. When paired with the Credo enhancements to migrate scheduling context upon invocation, synchronous IPC becomes quite similar to thread migration indeed. The desired behavior seems better captured by thread migration.

Predictable IPC execution time: In systems where it is difficult to predict the maximum number of threads on a wait-queue for a server thread (thus the worst-case cost of an IPC), it is possible for intelligent data-structures to provide a constant-time lookup of the highest priority thread waiting for IPC. This removes the linear increase to the cost of IPC for waiting threads (though not the cost commensurate with the depth of the dependency graph in Credo). The O(1) Linux scheduler (present in Linux versions 2.6 to 2.6.23) includes a data-structure enabling constant time lookup of the highest-priority thread. Though this approach will technically make IPC time predictable across all wait-queue lengths, it could impose a large cost in terms of memory usage and constant execution overheads. We leave this as an area of future study.

5 System Configurability

We discuss the ways that COMPOSITE provides the user-level definition of novel policies that rely upon the semantics of thread migration. Specifically, we discuss user-level, component-based scheduling, and Mutable Protection Domains (MPD) that enable the alteration of the protection domain configuration at run-time.

5.1 Component-Based Scheduling

In designing μ-kernels and component-based operating systems, a common goal is to include in the kernel only those concepts required to implement the system's required functionality at user-level [11]. The inclination is to remove mechanisms and policy from the (fixed) kernel and define them instead in replaceable and independently failable user-level components. Part of the motivation for this is so that the system can be configured to the largest possible breadth of application and system requirements. In real-time and embedded systems, the policies that dictate temporal behavior are amongst the most sensitive to meeting such requirements. In COMPOSITE, then, we have focuses on enabling the user-level, component-based definition of system scheduling policies [13].

To enable efficient user-level scheduling in COMPOSITE, the invocation path should not require scheduler invocation. This goal has two implications: First, the invocation path should not rely on scheduling parameters associated with threads, such as priority, as these are defined in the user-level scheduler. Second, the invocation path should not result in multiple threads becoming active, as this would imply an invocation of the user-level scheduler.

The migrating thread model satisfies both of these constraints. As no thread switches occur during the invocation path, the scheduling parameters associated with threads are not required. The only thread active during an invocation is the original scheduling context. To block or wakeup threads, invocations must be made to the scheduler component.

Pure synchronous IPC does not satisfy either of these goals. As thread switches occur on each IPC, the next thread to execute must be located, and to do so involves access to thread scheduling parameters, and dispatching between threads. This practically requires the scheduler to be kernel-resident, and for the IPC mechanism to hard-code a single scheduling policy. Additionally, some IPC operations result in the activation of multiple threads. For example, when executing a *reply_wait*, the IPC path can result in both the client and server threads, being active if there are threads on the wait-queue for the server thread. Direct process switching avoids these issues at the cost of predictable thread execution accounting.

There are some indications, beyond the COMPOSITE implementation, that user-level scheduling of all system threads is best done with the migrating thread model. For example, in [21], L4 is modified to allow specific threads to control scheduling. Doing so involves migrating scheduling context with IPCs as in Credo. Additionally, due to complications created by end-point contention, the author suggests that a solution is to "construct a μ-kernel solely based on procedure call semantics".

5.2 Mutable Protection Domains

The resource accountability and execution semantics of component invocations are identical for invocations between protection domains, and between components in the same protection domain. This, along with novel mechanisms for predictably and dynamically altering protection domain structures, enables Mutable Protection Domains (MPD) [12]. MPDs recognizes that in a fine-grained component-based system, even optimized invocation paths can have significant overheads[10]. The system is able to monitor the frequency of invocations between each component. We observe that the distribution of such invocation counts is heavily tailed, and if the overhead of invocations between a small number of components is removed, the system can attain both high reliability (retaining most protection domain boundaries), while concurrently achieving significant performance improvements (up to 40%). As the distribution of invocations between components change, the system can erect and remove protection boundaries as appropriate. The goal is to maintain high reliability (to detect and isolate faults when they occur), and high performance. When a protection boundary is required for security, it should never be removed.

To retain a consistent model of thread execution accounting and scheduling in a system using synchronous IPC, thread switches would be necessary even when the components share a protection domain. The overhead of scheduling and switching between threads is higher than that of direct invocation of the destination function. For example, the cost of intra-protection domain invocations in COMPOSITE

[10]In COMPOSITE, we implement a simple web-server [14] consisting of about 25 components. Each HTTP request causes between 50 and 70 invocations depending on if it is for static or dynamic content.

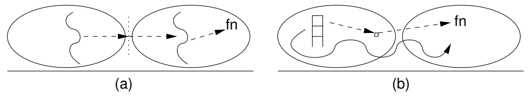

Figure 7. Inter-address space invocations of the function `fn`, using (a) synchronous IPC that required thread dispatch, and (b) migrating threads with indirect invocations through a function pointer [14].

is on the order of a C++ virtual function call, significantly faster than thread dispatch. Figure 7 depicts these two forms of invocation. Only a single execution context is required when using thread migration.

6 Thread Migration Limitations

There are a number of limitations and peculiarities both with the thread migration model, and with the COMPOSITE implementation. We discuss these in turn.

Execution Context Unavailability: When an invocation is made to a component, an execution context for that invocation must be found. This can be done in the kernel [8], or within the component itself (as in COMPOSITE). If there are no available contexts, the system must resolve this contention. Both thread migration *and* synchronous IPC must deal with this case where there are more pending invocations than there are server execution contexts. We believe this case is best dealt with by allowing the customized definition of the policies for dealing with such cases. In COMPOSITE, these policies are defined as components and they vary from having separate execution stack freelists for different client service classes (i.e. to guarantee that hard real-time tasks will always find a stack), to implementing priority inheritance. Additionally, we are currently investigating methods to balance responsiveness with execution context memory usage when allocating execution contexts to components.

Fault Recovery: The fault recovery model for server-based systems using synchronous IPC is well-known and simple to understand. When a fault occurs within a server, all threads in active IPC with that thread or server can be directly notified of that failure and act accordingly. The faulting thread and the server can independently be restarted. With thread migration, the thread that causes a fault in one component should not be destroyed as its execution context is spread across multiple components. Thus when a fault occurs in a specific component, one solution is to cause the thread to return to the invoking component, either with an error code, or an exception [3]. This model is less familiar to developers accustomed to a process-style structuring of the system.

6.1 COMPOSITE Implementation Limitations

COMPOSITE is a prototype and should not be seen as being as feature-rich as either monolithic systems such as Linux, or even mature μ-kernels such as L4. A number of design decisions simplify the implementation of thread migration in COMPOSITE.

First, the COMPOSITE kernel is non-preemptive. The IPC path assumes that no interrupts will preempt it, thus that no synchronization around kernel data-structures is required in the single-processor case. Additionally, as the invocation path does not touch user-memory, we assume that faults cannot occur. The non-preemptive assumption is justified by the general lack of expensive operations in the kernel. For example, we avoid supporting general hierarchical address space operations such as *map*, *grant*, and *unmap* [11] operations in the kernel, as complex mapping hierarchies can cause *unmap* to become expensive. Instead, we provide a simple operation to directly map a physical frame into a specific virtual location in a component. A privileged user-level component uses this simple facility to itself implement the higher-level operations. Though we believe the non-preemptive kernel implementation is a sound design decision, we cannot predict if systems that do not make such an assumption might have difficulty implementing efficient invocations using thread migration.

An additional limitation of the current COMPOSITE implementation is the fact that it only supports uniprocessors. We believe partitioning the state of the system (e.g. threads) between processors will enable efficient and predictable (and lock-free) invocations when we move COMPOSITE to multiprocessors.

7 Related Work

Thread migration is not a new method for inter-protection domain invocation. LRPC [1] describes now RPC within a single machine can be optimized by using thread migration. Ford [7] altered the invocation path in Mach to use thread migration for a significant performance improvement. Pebble [8] optimizes invocation latency by custom-compiling specialized invocation code and by using thread migration. We argue that thread migration is a predictable foundation upon which to implement finely decomposed, configurable, and reliable systems. We do not know of other work that has compared the predictability of thread migration to synchronous IPC.

8 Conclusions

In this paper, we argue the case for using a thread migration approach for predictable inter-protection domain communication in configurable and reliable systems. In doing so, we introduce the COMPOSITE design for component invocation that uses thread migration. We make this argument in terms of three factors: (1) the desire to have a consistent processor management and accounting scheme for CPU utilization across invocations that maps well to systems in which specialized services are provided by some components to others, (2) the communication end-point abstractions provided by the kernel that have a significant effect on the bounds for IPC latency, and (3) the effect that the IPC mechanism can have on the ability of the system to provide configurable system policies (e.g. scheduling, MPD).

We argue that thread migration provides a predictable invocation foundation, and we contrast that with synchronous IPC in the general case. We argue not that previous IPC mechanisms should be abandoned, but that bringing their semantics closer to that of thread migration is beneficial for overall system predictability.

The COMPOSITE source code is available upon request.

References

[1] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, 1990.

[2] B. N. Bershad, D. D. Redell, and J. R. Ellis. Fast mutual exclusion for uniprocessors. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 223–233, New York, NY, USA, 1992. ACM.

[3] F. M. David, J. C. Carlyle, E. Chan, D. Raila, and R. H. Campbell. Exception handling in the choices operating system. In *Advanced Topics in Exception Handling Techniques in Springer Lecture Notes in Computer Science*, pages 42–61, 2006.

[4] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: a flexible, optimizing idl compiler. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 44–56, New York, NY, USA, 1997. ACM Press.

[5] K. Elphinstone. Resources and priorities. In *Proceedings of the 2nd Workshop on Microkernels and Microkernel-Based Systems*, October 2001.

[6] K. Elphinstone, D. Greenaway, and S. Ruocco. Lazy scheduling and direct process switch – merit or myths? In *Workshop on Operating System Platforms for Embedded Real-Time Applications*, July 2007.

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

[8] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The pebble component-based operating system. In *Proceedings of Usenix Annual Technical Conference*, pages 267–282, June 2002.

[9] A. Lackorzynski and A. Warg. Taming subsystems: capabilities as universal resource access control in l4. In *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, pages 25–30, New York, NY, USA, 2009. ACM.

[10] J. Liedtke. Improving ipc by kernel design. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 175–188, New York, NY, USA, 1993. ACM Press.

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

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

[13] 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.

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

[15] S. Reichelt, J. Stoess, and F. Bellosa. A microkernel api for fine-grained decomposition. In *5th ACM SIGOPS Workshop on Programming Languages and Operating Systems (PLOS 2009)*, Big Sky, Montana, oct 2009.

[16] S. Ruocco. A real-time programmer's tour of general-purpose l4 microkernels. In *EURASIP Journal on Embedded Systems*, 2008.

[17] Scheduling in k42, whitepaper: http://www.research.ibm.com/k42/white-papers/scheduling.pdf.

[18] 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.

[19] J. S. Shapiro. Vulnerabilities in synchronous ipc designs. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 251, Washington, DC, USA, 2003. IEEE Computer Society.

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

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

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

[23] V. Uhlig, U. Dannowski, E. Skoglund, A. Haeberlen, and G. Heiser. Performance of address-space multiplexing on the Pentium. Technical Report 2002-1, University of Karlsruhe, Germany, 2002.