BOSTON UNIVERSITY
GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

COMPOSITE : A COMPONENT-BASED OPERATING SYSTEM FOR PREDICTABLE AND DEPENDABLE COMPUTING

by

GABRIEL AMMON PARMER
B.A., Boston University, 2003

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
2010

Approved by

First Reader

Richard West, Ph.D
Associate Professor of Computer Science

Second Reader

Azer Bestavros, Ph.D
Professor of Computer Science

Third Reader

Ibrahim Matta, Ph.D
Associate Professor of Computer Science

Acknowledgments

This thesis is the product not only of my own will, but also of the generosity and passion of many others who aided my intellectual development.

This thesis would not be possible, on any level, were it not for my adviser, Professor West. He has exhibited an amazing combination of traits that have fostered our symbiotic relationship. I can only feel immense gratitude for the years of patience and help he gave me as I acquired the knowledge to be productive. During this development, he pushed me in directions far outside of my comfort zone, and despite my stubborn nature and complaints, he was able to turn me into a far better researcher. Professor West knew when to take a leap of faith and let me chase the crazy idea of a component-based OS. Without being given this freedom to work on my passion, this thesis would not have come to be. Finally, I want to thank Rich for being not only my adviser, but a great friend.

Numerous professors have contributed to the quality and breadth of this thesis by shifting my manner of thought. Professor Bestavros' manner of thinking about and formulating a new problem or solution is illuminating and something I can only hope to emulate. His contributions to the department, to the Systems group, and to this thesis are deeply valued and appreciated. Professors Kfoury and Xi by offering courses, seminars, and group meetings gave me the opportunity to delve much deeper into formal methods and programming languages than I otherwise would have been able. Their patience in delivering difficult concepts has always been appreciated. Professor Teng always provides a jovial sense of solidarity late at night in the lab; I will always be happy to water his plants.

During my long tenure at Boston University, I've come to increasingly appreciate the environment and influences that stimulated thought and fostered my intellectual progres-

sion. I cannot thank those professors enough who brought a passionate desire to their craft of passing on knowledge in CS, Math, and Philosophy. They helped animate my desire to pass on my own knowledge. Additionally, the impact of my pre-college educators cannot be overstated. Many of them went beyond their designated duties and provided an environment in which self-challenge and aspiration were not only possible, but encouraged. I specifically wish to thank Diane Catron and Anita Gerlach for, in very different ways, vitalizing my intellectual and scholarly cravings.

Many friends, scattered across the country, have contributed to this thesis. I'm very lucky to have gifted, successful, and inspirational friends that lightened my workload by expanding, by proxy, the breadth of my experiences and happiness. It has been a joy to grow older with them, and to grow richer through them. At BU, Michael Ocean is an invaluable sounding-board and, quite simply, an absolute pleasure to be around. We took on this adventure together, and I am better for it. Ernest Kim is always able to focus a practical lens on life and on my research that is both refreshing and valuable. His compassion has helped me survive the graduate-student lifestyle that was sometimes trying.

My Dad and Mom have always exhibited an unconditional support for my choices and directions. Without the opportunities they made available to me, often at their own expense, this thesis would not exist. They gave me a strong foundation upon which to build, and in times of difficulty, gave me a foundation on which to lean and rest. This thesis is as much a remuneration of their work and effort as it is of mine.

Few events in my life have caused a tectonic shift in my perception, philosophy, and happiness. Kimberly Gerson provided the largest of these, and it is because of her that my notions of quality of life have been completely redefined to a much higher standard. Her dedication to her passions is an inspiration and has reinforced my own will to accomplish the work behind this thesis. It is with her that I undertook this challenge, and with her we will meet those challenges to come.

COMPOSITE : A COMPONENT-BASED OPERATING SYSTEM FOR PREDICTABLE AND DEPENDABLE COMPUTING

(Order No.)

GABRIEL AMMON PARMER

Boston University, Graduate School of Arts and Sciences, 2010

Major Professor: Richard West, Associate Professor of Computer Science

ABSTRACT

Systems in general, and embedded systems in particular, are increasing in software complexity. This trend will only continue as we expect more functionality from our computational infrastructure. With complexity, the system's ability to tolerate and be resilient to faulty or malicious software becomes ever more challenging. Additionally, as system capabilities increase, it becomes impossible for the operating system (OS) designer to predict the policies, abstractions, and mechanisms required by all possible applications. These trends motivate a system architecture that places an emphasis on both dependability and extensibility.

This thesis presents the COMPOSITE component-based OS that focuses on system-provided fault tolerance and application-specific system composition. A goal of this system is to define resource management policies and abstractions as replaceable user-level components. Importantly, this enables the component-based control of both the temporal- and memory-isolation properties of the system. All system scheduling decisions are component-defined, as are policies that determine the configuration of fault-isolation barriers throughout the system. In achieving this goal, we posit a philosophy in which fault-isolation is not a binary condition (that is, present or not), but rather dynamically controlled by the system's components.

This thesis first focuses on how COMPOSITE is able to migrate the system CPU scheduling policy implementation from the trusted kernel to user-space component. In this way, scheduling policy is application-specific and fault-isolated from other components. We demonstrate how different component-defined policies for controlling temporal aspects of

the system are able to predictably schedule interrupt execution to prevent livelock.

The second main focus of this thesis is an investigation of the trade-off between fault-isolation and system performance. Protection domains between components provide fault-isolation, but inter-protection domain communication incurs a performance overhead. In recognition of this trade-off, we introduce Mutable Protection Domains, a novel mechanism to dynamically construct and remove isolation boundaries within the system in response to changing inter-protection domain communication overheads. Using this mechanism, we demonstrate that a component-based web-server is able to manipulate its protection domain configuration to achieve throughput improvements of up to 40% over a static configuration while concurrently maintaining high fault isolation.

Contents

List of Tables

List of Figures

List of Abbreviations

API . . . . . Application Programming Interface

CBOS . . . . . Component-Based Operation System

CGI . . . . . Common Gateway Interface

COTS . . . . . Commodity Off The Shelf

CPU . . . . . Central Processing Unit

HTTP . . . . . Hyper-Text Transfer Protocol

IP . . . . . Internet Protocol

IPC . . . . . Inter-Process Communication

MMKP . . . . . Multiple-choice, Multi-dimensional Knapsack Problem

MPD . . . . . Mutable Protection Domains

NIC . . . . . Network Interface Card

OS . . . . . Operating System

RAS . . . . . Restartable Atomic Sequences

RPC . . . . . Remote Procedure Call

TCP . . . . . Transmission Control Protocol

VM . . . . . Virtual Machine

VMM . . . . . Virtual Machine Monitor

WCET . . . . . Worst-Case Execution Time

Chapter 1

Introduction and Motivation

1.1 Motivation

Computer systems, and especially embedded systems, are undergoing a steady rise in software complexity. The functionality required by modern cell phones is increasing to the point of matching desktop software suites. Embedded systems are increasingly hybrid systems, or those with tasks of mixed temporal constraints, and are being used with sensors and actuators for relatively unpredictable physical processes.

1.1.1 System Dependability

Both the increasing complexity of the software deployed on systems and the sometimes disastrous consequences of system failures motivate a serious effort towards ensuring the dependability of such systems. Especially in the embedded domain, the heightened complexity poses an acute problem: systems that could be verified to a high degree either statically or through extensive testing are pushing the boundaries of validation procedures. Examples of systems that went through a thorough validation process yet still failed include (i) the NASA mars orbiter [Doub] , (ii) Ariane 5 [Doua], a rocket designed to transfer payloads to orbit with a budget in the billions, and (iii) the electrical control systems that failed causing the east-coast blackout of 2003 [Pou]. Indeed, software failures and system failures

are a wide-spread problem. The US National Institute of Standards and Technology (NIST) found in 2002 that software bugs cost the US economy $59.5 billion dollars annually [NIS]. System failures cause monetary damage or, in the extreme case, the loss of life and systems should be designed to prevent the scope of failures. Toward this goal of increasing system reliability, high-confidence systems often rely on a combination of fault prevention and fault tolerance [LR04] to minimize the probability of system failure. Most operating systems in common use, however, do not promote a systematic process for fine-grained fault isolation either within applications themselves, or within the run-time system's code. Instead, if an application failure is detected, the entire application is generally terminated, or if a kernel fault is found, the system must be rebooted to recover from a possibly inconsistent state. This coarse-grained fault isolation hampers the ability of the system to recover on a granularity smaller than an application.

1.1.2 System Predictability

In addition to dependability requirements, the correctness of real-time systems is dependent on the system's ability to meet application temporal constraints. Increasingly complex systems and applications stress the system's ability to meet these constraints. Expressed in terms of Quality of Service or tasks deadlines, applications' resource requirements define the service levels required from the system. To behave predictably and support the correctness of these applications, the system must contain resource management policies specialized towards specific application temporal constraints. For this goal to be achievable across all systems and application scenarios, the system policy for controlling CPU allocation, the scheduler, must be customizable and provide exactly those resource management services required by the system.

1.1.3 System Extensibility

In spite of growing software complexity, there is little unification of the run-time systems that enable a wide variety of these application domains. Different operating systems provide

different trade-offs between throughput and latency, between scalability and simplicity, and between performance and memory usage, so it is not a surprise that the run-time system in a deeply embedded system is different than that of a cluster-computing node. Even amongst applications within the same domain, a single system often does not provide the optimal interface or policies for all applications. The ideal page-replacement policies for a database [Sto81] or a garbage-collected runtime [HFB05] are different than those for general purpose applications, and the ideal scheduling policy for an event-triggered real-time process is different than for one that is time-triggered. In short, systems often exhibit a *semantic gap*, or the difference between the system requirements of applications and the abstractions and policies provided by the run-time system. Instead of systematically approaching this problem and making a single system that scales across all problem domains by bridging the semantic gap, there is a multitude of run-time systems each tailored to a domain, with little code-reuse between them.

This thesis seeks to address this issue by detailing a system that provides extensibility and application-specific composition of the system's policies and abstractions. In supporting predictability across the widest breadth of application-domains we provide customizable scheduling policies. More fundamentally, system customizability is explored as a first-class priority to bridge the semantic gap.

1.2 The Composite Component-Based OS

This thesis centers around the design, implementation, and evaluation of the Composite component-based operating system (CBOS). A component in this context is a collection of code and data that provides functionality in accordance to a contractually-specified interface. Each component is independently redeployable and harnesses the functionality of other components through their interfaces. A set of components is composed into a functional system. The decoupling of interface and implementation enables specifically those policies and abstractions to be used in a system that provide behavior towards the system's

or application's goals. This design bridges the semantic gap, thus making COMPOSITE applicable to domains ranging from constrained embedded systems to servers. Essentially, COMPOSITE is a common architecture on which complicated and specialized behaviors are composed from a set of common components.

Segregating the system into components provides the opportunity for increased system fault isolation as each component is placed into its own hardware-provided protection domain at user-level (implemented via *e.g.* page-tables). A fault due to malicious or erroneous code in any component is prevented from trivially propagating to and corrupting the memory of other components or the trusted kernel. Additionally, protection domains provide a natural encapsulation of state for a component. If a component exhibits a failure, its code and data can be independently restarted avoiding a full-system reboot [CKF+04]. Unfortunately, switching between hardware protection domains is expensive relative to a direct function call. This is mainly due to overheads in crossing between protection levels, and the necessary invalidation of hardware (virtually indexed) caches. With each component in a separate protection domain, communication between them necessitates these switches, and imposes significant overhead on the system.

1.2.1 Predictable Component-Based Scheduling

The goal of CBOSes is to enable the component-based definition of *all* system policies that might effect an application's ability to meet its goals. Component-based scheduling is important as it allows application-specific specialization of system scheduling policies. This is motivated by the fact that different application goals and constraints are ideally solved by different scheduling policies [vBCZ+03, Ruo06]. As applications do not generally trust each other to manage all system resources, these policies must be defined at user-level in a manner that isolates their scheduling decisions. However, past systems have shown that scheduling policies are difficult to migrate to a user-level, untrusted, component-based implementation. Schedulers are involved in many performance-critical execution paths, thus overall system performance is sensitive to the cost of their invocation. The main challenges

associated with providing component-based schedulers include:

(1) Schedulers often require synchronization (around structures shared across threads, *e.g.* the runqueue) which generally implies a kernel-provided mechanism to control access to critical sections (e.g. semaphores). Unfortunately, such a mechanism requires scheduler support to switch between threads. This circular dependency between kernel and component scheduler must be resolved for a plausible user-level scheduling solution.

(2) The processing that results from hardware asynchronous events (*i.e.* interrupts) must be scheduled in much the same manner as normal threads. Naively, this implies scheduler invocations for each interrupt. However, interrupts often execute with an high frequency. To invoke a scheduler (at user-level in a separate protection domain) for each interrupt only increases overheads that lead to livelock if not controlled properly [MR97, RD05].

In embedded and real-time systems, the policies that define the temporal characteristics of the system directly influence the correctness of the system. For a system to support the timing behavior of tasks and interrupts that is ideal for all real-time applications, a system must allow the scheduling policies to vary according to system and application goals.

1.2.2 Mutable Protection Domains

COMPOSITE places a systematic emphasis on fault-isolation. All components are executed at user-level in separate protection domains such that they cannot maliciously or accidentally modify the data-structures, or alter the execution state of other components. The isolation boundaries separating components have two implications:

(i) When a failure (such as data-structure corruption, or deadlock) occurs in one component, the scope of the failure is contained to that component. The recovery from inevitable failure of a component only necessitates restarting that specific component.

(ii) Communication between components must be actuated by the kernel as only the kernel has the permission to switch protection domains. These invocations impose significant overhead beyond a simple function call. Unfortunately these overheads prevent

some applications from meeting performance or predictability constraints, depending on the inter-component communication patterns and overheads of the system.

A significant contribution of this thesis is the identification of utility in allowing the system to dynamically leverage the trade-off between the granularity of fault isolation, and the performance of the system. We propose and investigate Mutable Protection Domains (MPD) which allows protection domain boundaries to be erected and removed dynamically as the performance bottlenecks of the system change. When there are large communication overheads between two components due to protection domain switches, the protection domain boundary is removed, if necessary. In areas of the system where protection domain boundaries have been removed, but there is little inter-component communication, boundaries are reinstated.

The ability to dynamically alter the protection domain configuration of the system is important as it is difficult to predict the bottlenecks of an active system, thus the ability of the system to automatically find them is useful. In complex systems with varying workloads, those bottlenecks can change over time. In such systems, the ability to dynamically adapt to communication patterns between components is essential to maintain consistently high performance (*e.g.* in terms of latency or throughput), and high fault-isolation properties. Importantly, each system places different importance on reliability and performance. MPD allows each system to make the trade-off that best fits their purpose. This thesis reports the design, implementation, and evaluation of the COMPOSITE MPD mechanisms. Additionally, we investigate policies that can be used to determine where protection domains should be placed given inter-component communication overheads.

1.3 Thesis Organization

What follows is an overview of the chapters of this thesis.

Chapter 2: Related Work

This chapter surveys past research related to COMPOSITE. Specifically, we outline relevant work on component-based OSes, component-based scheduling, and OS fault-isolation.

Chapter 3: Component-Based Scheduling

This chapter investigates how system scheduling can be provided as a user-level component. This allows the scheduling policy itself to be application-specific and fault-isolated from the rest of the system. Component-based scheduling raises a number of complications involving synchronization, interactions with component invocation, and efficiently controlling interrupt execution.

Chapter 4: Mutable Protection Domains: Policy

This chapter details a model for a component-based system that includes both overheads between components, and a notion of dynamic protection domains. Using this model, we formulate how a policy is constructed to, given overheads for communication between components, compute where protection domains should be present in the system. Included is an investigation into the algorithmic and analytical aspects of system support for MPD.

Chapter 5: Mutable Protection Domains: Design and Implementation

This chapter delves into the mechanisms concerning how COMPOSITE is designed and implemented to support MPD. This investigation includes not only providing abstractions to the policy component to control the protection domain configuration, but also ensuring these mechanisms do not impede other critical subsystems.

Chapter 6: Future Work

Here we describe outstanding issues and directions for further investigation.

Chapter 7: Conclusion

In this final chapter, we conclude this thesis with some closing remarks.

Chapter 2

Related Work

2.1 System Extensibility

Composite bridges the semantic gap by allowing the definition of system policies and abstractions as specific components that are combined to yield a system tailored to the application domain. Past research projects have attempted to span the semantic gap using different mechanisms. A discussion of these follows.

2.1.1 User-Level Sandboxing

User-Level Sandboxing [WP06] (ULS) focuses on providing an execution environment in Commodity Off the Shelf (COTS) systems in which trusted application-specific extensions are run. The kernel of the system is isolated from such extensions, yet it is possible to execute them in a variety of execution contexts including at interrupt time. This enabled our past research into customizable networking [QPW04]. ULS allows the system to recover from erroneous extensions, but makes no attempt to fundamentally make the system as a whole (excluding extensions) more reliable, or to redefine the base system to make it extensible to all application-domains.

2.1.2 Hijack

Our previous work on Hijack [PW07a] enables the redefinition of the services provided by a base COTS system for a set of processes by interposing on their service requests. The interposition code is isolated via hardware from the applications, and the kernel is isolated from the interposition logic. Interposition code is still able to harness the services of the base COTS system, and, were appropriate, to redefine policies in an application-specific manner. This technique bridges the semantic gap for application requests, but does not increase the isolation characteristics of applications, interposition code, or of the kernel.

2.1.3 Modules/Drivers

One common solution to the semantic gap is to provide a mechanism to download code directly into a running kernel. Systems such as Linux [Lin] provide this ability through modules. This approach is limited in that modules only interface with the kernel at statically defined locations, and they cannot be used to override base policies. For example, the pseudo-LRU policies of the page-cache, or the CPU scheduler cannot be replaced. More significantly, modules are run at kernel-level and an error in one trivially propagates into the kernel-proper. This, therefore, is a mechanism that is only used with highly trusted extensions, and that does not promote system fault-tolerance.

2.1.4 Exokernels

Exokernels [EKO95, KEG+97] represent a different design philosophy where most system functionality is placed in the same protection domain as the applications themselves in libraries. This has the effect of bridging the semantic gap as system policies loaded as library OSes are chosen and written specifically by the applications themselves. Unfortunately, faults that exist anywhere in a single application will require restarting the entire application and its library OSes. On embedded systems with a limited number of applications executing, this is equivalent to total system failure. Fault-tolerance is not emphasized throughout the entire software stack. Additionally, in exokernels any policies for managing

resources that must be defined in a central location, are placed into a separate protection domain and invoked via expensive inter-process communication. For certain applications, the overhead of this communication can compromise temporal constraints. Managing the trade-off between fault isolation and system performance is the focus of MPD.

2.1.5 Vino and Spin

Vino [SESS96] and Spin [BSP+95] attempt to bridge the semantic gap by combining the ability of modules to alter the base system using downloaded code, and software techniques to protect against extension failure. Specifically, code downloaded into the kernel is disallowed from accessing arbitrary regions in memory to protect the kernel using either type safe languages, or software fault isolation [RWG93]. In Vino, when an extension is invoked, it is executed as a transaction that is only committed when the invocation completes. This prevents kernel data-structures from being corrupted if an extension fails in the middle of an invocation. The approaches of Spin and Vino are limited because, as with modules, extensions can only implement policies at statically-defined extension points. Importantly, fine-grained fault isolation within applications is not a focus of either work as their extensibility mechanisms focus on the kernel. Additionally, any overheads associated with software protection schemes [LES+97] cannot be dynamically removed, and might preclude meeting application performance goals.

2.2 System-Provided Fault Isolation

MPD is a novel construct in COMPOSITE enabling the system to trade-off fault isolation for efficiency. A focus on fault tolerance has been seen for some time as essential to dependable systems [LR04]. Here we survey past OS research with a focus on system provided fault-tolerance.

2.2.1 μ-kernels

μ-kernels [Lie95, FAH+06, SSF99, KAR+06] structure the system such that functionality that would otherwise be defined in the kernel instead exists in servers at user-level. This design is similar to COMPOSITE in that the policies of the system are defined in separate protection domains at user-level, thus promoting fault-isolation. However, protection domain boundaries are defined statically in the system. Care must be taken when choosing where to place them relative to the software components of the system. Protection domains must be placed such that they don't induce overheads for *any* execution pattern that could preclude meeting application performance requirements. Because of this, protection domains must be placed conservatively, with the performance of all possible flows of control in mind. This contrasts with MPD that attempts to place protection domain boundaries throughout the system in an application- and execution-specific manner that takes system and application performance constraints into account while attempting to maximize system fault isolation. Because user-level servers can be replaced in an application-specific manner, there is some possibility for bridging the semantic gap. However, this can only be done on the granularity of a server, not on that of individual policies within a server. Additionally, existing μ-kernels do not allow the efficient user-level untrusted component-based redefinition of system temporal policies (including the scheduling policy).

2.2.2 Nooks

Nooks [SBL03, SABL04] specifically attempts to provide increased fault-isolation for drivers in legacy systems such as Linux. This is achieved by using hardware techniques to isolate drivers from the base kernel while still executing them at kernel-level. The focus here is to prevent accidental failures and little protection is afforded against malicious drivers. Unlike Nooks, legacy code support is not a primary focus of COMPOSITE. Instead, COMPOSITE provides pervasive fault isolation boundaries throughout all components of the system.

2.3 Component-Based Operating Systems

Components bridge the semantic gap by allowing for application-specific system composition. The use of components has been shown to be useful in very specialized systems such as Click [MKJK99], where complicated routing policies can be constructed from components to yield a high-performance, customized networking appliance. Components have also scaled up to server middleware to provide application-processing layers for web requests [Ent], or distributed coordination [JAU]. Here we focus on component-based OSes [FSH+01] that provide both the ability to implement low-level policies such as networking and memory management, but also application-level logic.

2.3.1 TinyOS

TinyOS [HSW+00] provides an execution environment for applications running on constrainted hardware such as sensor motes. System services are defined as components and they are pieced together to yield a functional system. The development style and approach to code-reuse is similar to Composite, but because of the hardware TinyOS focuses on, there is no hardware-provided fault isolation. Indeed the entire system is simplified due to hardware constraints, to the point of not encouraging the use of full threads. This approach, therefore does not scale past the simple applications targeted to the sensor motes.

2.3.2 Pebble

Pebble [GSB+02] is an operating system that also encourages system development from user-level components in separate protection domains. The focus of this work is on the optimization of the communication mechanism between components. When components wish to communicate, code is dynamically generated with aggressive inlining of kernel data-structure addresses and specialization to this specific execution path. The goal is to perform the least amount of operations possible to yield proper communication, thus resulting in very efficient communication. Indeed, inspiration is taken from this work for Composite.

Invocations between components in the same protection domain (due to a protection barrier being removed with MPD) are optimized by dynamically generating the invocation code. Pebble is unique amongst previous work in that it does allow the definition of the system's scheduler in a user-level component. Unfortunately, these schedulers must be trusted as they are allowed to disable system interrupts for synchronization. An errant scheduler could trivially halt execution of any useful code on the machine by infinite looping with interrupts disabled. Additionally, Pebble must invoke the scheduler for each interrupt, resulting in significant overheard. Pebble is implemented on a processor with low kernel/user level switches, and fast protection domain switches. By comparison, COMPOSITE targets a commodity processor without specialized features such as address-space ids that significantly decrease the processing cost of switching protection domains, and must address the overheads that result. Pebble shares many of the disadvantages of μ-kernel such as a static system structure that does not adapt to changing inter-component communication patterns.

2.3.3 Think and the Flux OS Kit

Think [FSLM02], the Flux OS Kit [FHL+96], and CAMKES [KLGH07] focus on the component-based design of operating systems. In these systems, a designer not only chooses which components the system is to consist of, but also the manner in which they communicate with each other and the placement of protection domains. This allows an additional dimension of configurability above μ-kernels in that the notion of a component is decoupled from that of a protection domain. In these systems, system structure is static and cannot adapt to run-time changes in communication bottlenecks as in COMPOSITE.

2.4 Extensible Scheduling Policy

Application-specific scheduling policies are useful in a wide variety of contexts, from scientific computing [ABLL91], to real-time scheduling [Ruo06], to serving webpages [vBCZ+03]. The approaches for providing extensible scheduling can be classified into three categories:

(1) $N : M$ techniques in which N user-level threads are multiplexed onto M system-level threads, (2) implementations that allow the alteration of a scheduler hierarchy in the kernel, and (3) approaches similar to COMPOSITE that allow the control of system-level threads via a user-level scheduler. Here we survey each alternative.

2.4.1 $N : M$ Thread Systems

User-level thread operations such as dispatching and coordination are inherently less expensive than system-level (kernel-provided) alternatives. The primary observation is that the kernel operations require a user-kernel protection level switch, which is relatively expensive [Ous90], while the user-level equivalents do not. Unfortunately, when a user-level thread makes a blocking I/O request, it will block, starving the execution of the rest of the user-threads. Schemes permitting coordination between user- and system-level threads [ABLL91, MSLM91, vBCZ+03, Sch] are able to achieve many of the benefits of user-level threads by permitting efficient inter-thread operations, while still inter-operating with the rest of the system (notably I/O). Unfortunately, such approaches only alter the scheduling behavior for a single process's threads, and do not alter the base policies for scheduling system threads, including low-level execution paths for processing I/O (*e.g.*`softirq`s in Linux). If the system's scheduler makes it impossible to satisfy an application's requirements, user-level threads are not beneficial.

Virtual machines (VMs) [PG74] fall into this category. Within the VM, threads are managed according to the VM's scheduling policy. However, for the application within a VM to meet it's goals, it still dependents on the system scheduler (in the virtual machine monitor (VMM)) that controls all VMs.

2.4.2 Extensible Kernel Scheduler

A number of systems enable the modification of the system scheduler in the trusted kernel. Shark [GAGB01] provides an interface for adding application-specific schedulers to their kernel, and communicating Quality of Service (QoS) constraints from the application to

kernel. HLS [RS01] provides facilities to create a hierarchy of schedulers that can be chosen in an application specific manner within the kernel. Each of these approaches is able to schedule system-level threads, but they share the constraint that the scheduling policies are inserted directly into the most trusted kernel and must themselves be trusted. Practically, then, untrusted applications cannot have a direct effect on the system scheduler, as generally the system cannot insert their code into the kernel. Bossa [BM02] defines a domain-specific language for implementing schedulers in an event-driven manner. Specific scheduling policies are written in this language, and C code is generated that interfaces with the host system. The Bossa language provides safety guarantees at the cost of sacrificing generality. COMPOSITE instead allows schedulers implemented with general-purpose languages to execute in separate protection domains.

Both HLS and Bossa provide the opportunity to hierarchically arrange schedulers such that parent schedulers delegate the scheduling of threads to different child schedulers. This increases the flexibility of these systems, but does not remove the main limitations of requiring trusted schedulers (HLS) or constraining programmability (Bossa).

2.4.3 User-Level Control of System Threads

COMPOSITE is able to control system-level threads from user-level. A number of other research projects have the same goal. Pebble [GSB+02] enables user-level component-based schedulers as described in Section 2.3.2. Schedulers in Pebble, must still be trusted as they synchronize by disabling interrupts.

Past research has put forth mechanisms to implement hierarchically structured schedulers at user-level in separate protection domains [FS96, Sto07]. Additionally, others have made the argument that user-level scheduling is useful for real-time systems, and have provided methods accommodating it in a middleware setting [ANSG05]. None of these works attempt to remove all notions of blocking and scheduling from the kernel. Additionally, these approaches, do not provide a mechanism for scheduling and accounting asynchronous events (e.g., interrupts) without recourse to costly scheduler invocations. As these ap-

proaches require switches to scheduler threads, they also prohibit useful operations that increase the predictability of the system such as Mutable Protection Domains [PW07b].

Chapter 3

User-Level, Component-Based Scheduling

This chapter focuses on the design of predictable and efficient user-level scheduling in our COMPOSITE component-based system. In particular, we show how a hierarchy of component-based schedulers is supported with our system design. By isolating components in user-space in their own protection domain, we avoid potentially adverse interactions with the trusted kernel that could otherwise render the system inoperable, or could lead to unpredictability. Additionally, each component scheduler is isolated from the faults of other components in the system. We show how a series of schedulers can be composed to manage not only conventional threads of execution, but also the execution due to interrupts. Specifically, in situations where threads make I/O requests on devices that ultimately respond with interrupts, we ensure that interrupt handlers are scheduled in accordance with the urgency and importance of the threads that led to their occurrence. In essence, there is a *dependency* between interrupt and thread scheduling that is not adequately solved by many existing operating systems [DB96, ZW06], but which is addressed in our component-based system. Additionally, we investigate how critical sections are managed to ensure exclusive access to shared resources by multiple threads.

In the following sections we elaborate on the design details of component-based scheduling in COMPOSITE. Section 3.1 describes in further detail some of the design challenges of COMPOSITE, including the implementation of hierarchical schedulers. This is followed by

an experimental evaluation in Section 3.2. Section 3.3 concludes this chapter.

3.1 Composite Component-based Scheduling

In COMPOSITE, a system is constructed from a collection of user-level components that define the system's policies. These components communicate via thread migration [FL94] and compose to form a graph as depicted in Figure 3.1. Edges imply possible invocations. Here we show a simplified component graph with two tasks, a networking component, a fixed priority round-robin scheduler, and a deferrable server. As threads make component invocations, the system tracks their progress by maintaining an *invocation stack*, as shown in Figure 3.2. In this example, the downward control flow of a thread proceeds through `A`, `B`, `C` and `D`. Each of these invocations is reflected in its execution stack. On return from `D` and `C`, the thread pops components off of its execution stack, and invokes `E`. Its invocation stack after this action is shown with the dotted lines.

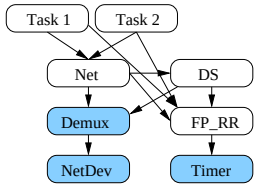

Figure 3.1: An example component graph.

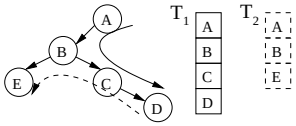

Figure 3.2: Thread execution through components.

One of the goals of COMPOSITE is to provide a base system that is configurable for the needs of individual applications. However, for performance isolation, it is necessary that global policies maintain system-wide service guarantees across all applications. Consequently, we employ a hierarchical scheduling scheme [RS01] whereby a series of successive

schedulers are composed in a manner that maintains control by more trusted schedulers, while still allowing policy specialization for individual applications. In addition to traditional notions of hierarchical scheduling, we require that parent schedulers do not trust their children, and are isolated from their effects. In this way, the affects of faulty or malicious schedulers are restricted to their subtree in the scheduling hierarchy. By comparison, scheduler activations [ABLL91] are based on the premise that user-level schedulers interact with a more trusted kernel scheduler in a manner that cannot subvert the kernel scheduler. COMPOSITE adopts a mechanism that generalizes this notion to a full hierarchy of schedulers that all exist at user-level.

COMPOSITE exports a system call API for controlling the scheduler hierarchy. This enables the construction of a recursive structure of schedulers whereby more trusted (parent) schedulers *grant* scheduling privileges to their children. Likewise, schedulers have the ability to *revoke*, transitively, all such scheduling permissions from their children. To bootstrap the system, one scheduler is chosen to be the root scheduler. Creating new child schedulers is done with feedback from abstract application requirements [GAGB01], or via application specified policies.

`COS_SCHED_PROMOTE_SCHED`	promote component to be a child scheduler
`COS_SCHED_DEMOTE_SCHED`	remove child subtree's schedulers privileges
`COS_SCHED_GRANT_THD`	grant scheduling privileges to child scheduler for a specific thread
`COS_SCHED_REVOKE_THD`	revoke scheduling privileges to a child scheduler's subtree for a specific thread
`COS_SCHED_SHARED_REGION`	specify a region to share with the kernel
`COS_SCHED_THD_EVT`	specify an event index in the shared region to associate with a thread

Table 3.1: `cos_sched_cntl` options.

A component that has been promoted to scheduler status has access to the `cos_sched_cntl(operation, thd_id, other)` system call. Here `operation` is simply a flag, the meaning of which is detailed in Table 3.1, and `other` is either a component id, or a location in memory, depending on the operation. In a hierarchy of scheduling compo-

nents only the root scheduler is allowed to create threads. This restriction prevents arbitrary schedulers from circumventing the root scheduler for allocating kernel thread structures. Thus, it is possible for the root scheduler to implement thread creation policies (e.g. quotas) for specific subsystems or applications. Threads are both created and passed up to other components via the `thd_id cos_thd_cntl (component_id, flags, arg1, arg2)` system call. To create a new thread, the root scheduler makes a system call with the `COS_THD_CREATE` flag. If a non-root scheduler attempts to create a new thread using this system call an error is returned. Threads all begin execution at a specific upcall function address added by a COMPOSITE library into the appropriate component. Such an invocation is passed three arguments: the reason for the upcall (in this case, because of thread creation) and the user-defined arguments `arg1` and `arg2`. These are used, for example, to emulate `pthread_create` by representing a function pointer and the argument to that function.

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

3.1.1 Implementing Component Schedulers

Unlike previous systems that provide user-level scheduling [FS96, ANSG05, Sto07], the operation of blocking in COMPOSITE is not built into the underlying kernel. This means that schedulers are able to provide customizable blocking semantics. Thus, it is possible for a scheduler to allow arbitrary blocking operations to time-out after waiting for a resource if, for example, a deadline is in jeopardy. In turn, user-level components may incorporate protocols

to combat priority inversion (e.g., priority inheritance or ceiling protocols [SRL90]).

Not only do schedulers define the blocking behavior, but also the policies to determine the relative importance of given threads over time. Given that schedulers are responsible for thread blocking and prioritization, the interesting question is what primitives does the kernel need to provide to allow the schedulers to have the greatest freedom in policy definition? In COMPOSITE, a single system call is provided, `cos_switch_thread(thd_id, flags)` that permits schedulers with sufficient permissions to dispatch a specific thread. This operation saves the current thread's registers into the corresponding kernel thread structure, and restores those of the thread referenced by `thd_id`. If the next thread was previously preempted, the current protection domain (i.e. page-table information) is switched to that of the component in which the thread is resident.

In COMPOSITE, each scheduling component in a hierarchy can be assigned a different degree of trust and, hence, different capabilities. This is related to scheduler activations, whereby the kernel scheduler is trusted by other services to provide blocking and waking functionality, and the user-level schedulers are notified of such events, but are not allowed the opportunity to control those blocked threads until they return into the less trusted domain. This designation of duties is imperative in sheltering more trusted schedulers from the potential ill-behavior of less trusted schedulers, increasing the reliability of the system. An example of why this is necessary follows: suppose a network device driver component requests the root scheduler to block a thread for a small amount of time, until the thread can begin transmission on a Time-Division Multiple-Access arbitrated channel. If a less trusted scheduler could then restart that thread before this period elapsed, it could cause detrimental contention on the channel. The delegation of blocking control to more trusted schedulers in the system must be supported when a hierarchy of schedulers is in operation. To allow more trusted schedulers to make resource contention decisions (such as blocking and waking) without being affected by less trusted schedulers, a flag is provided for the `cos_switch_thread` system call, `COS_STATE_SCHED_EXCL`, which implies that only the current scheduler and its parents are permitted to wake the thread that is being suspended.

3.1.2 Brands and Upcalls

COMPOSITE provides a notification mechanism to invoke components in response to asynchronous events. For example, components may be invoked in response to interrupts or events similar to signals in UNIX systems. In many systems, asynchronous events are handled in the context of the thread that is running at the time of the event occurrence. In such cases, care must be taken to ensure the asynchronous execution path is reentrant, or that it does not attempt to block on access to a lock that is currently being held by the interrupted thread. For this reason, asynchronous event notifications in COMPOSITE are handled in their own thread contexts, rather than on the stack of the thread that is active at the time of the event. Such threads have their own priorities so they may be scheduled in a uniform manner with other threads in the system. Additionally, a mechanism is needed to guide event notification through multiple components. For example, if a thread reads from a UDP socket, and an interrupt spawns an event notification, it may be necessary to traverse separate components that encompass both IP and UDP protocols.

Given these constraints, we introduce two concepts: *brands* and *upcalls*. A brand is a kernel structure that represents (1) a context, for the purposes of scheduling and accounting, and (2) an ordered sequence of components that are to be traversed during asynchronous event handling. Such brands have corresponding priorities that reflect the urgency and/or importance of handling a given event notification. An upcall is the active entity, or thread associated with a brand that actually executes the event notification. Brands and upcalls are created using the `cos_brand_cntl` system call. In the current implementation, only the root scheduler is allowed to make this system call as it involves creating threads. The system call takes a number of options to create a brand in a specific component, and to add upcalls to a brand. Scheduling permissions for brands and upcalls can be passed to child schedulers in the hierarchy in exactly the same fashion as with normal threads. An upcall associated with a given brand is invoked by the `cos_brand_upcall(brand_id, flags)` system call, in a component in which that brand has been created.

Figure 3.3 depicts branding and upcall execution. A thread traversing a specific path

of components, `A, B, C, D`, requests that a brand be created for invocation from `C`: T_B = `cos_brand_cntl(COS_BRAND_CREATE_BRAND, C)`. Thus, a brand is created that records the path already taken through components `A` and `B`. An upcall is added to this brand with `cos_brand_cntl(COS_BRAND_CREATE_UPCALL,` T_B`)`. When the upcall is executed, component `B` is invoked, as depicted with the coarsely dotted line. This example illustrates a subsequent component invocation from `B` to `E`, as depicted by the finely dotted line.

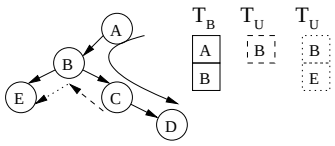

Figure 3.3: Branding and upcall execution.

When an upcall begins execution in a component, it invokes the generic upcall function added to the component via the COMPOSITE library. If an event occurs that requires the execution of an upcall for the same brand as an active upcall, there are two options. First, if there is an inactive upcall associated with a brand, then the inactive upcall can be executed immediately to process the event. The precise decision whether the upcall is immediately executed depends on the scheduling policy. Second, if all upcalls associated with a brand are active, then a brand's pending count of events is incremented. When an upcall completes execution and finds that its brand has a positive event count, the count is decremented and the upcall is re-instantiated.

Brands and upcalls in COMPOSITE satisfy the requirements for asynchronous event notification, but an important aspect is how to efficiently and predictably schedule their corresponding threads. When the execution of an upcall is attempted, a scheduling decision is required between the currently running thread and the upcall. The scheduler that makes this decision is the *closest common* scheduler in the hierarchy of both the upcall and the currently executing thread. Additionally, when an upcall has completed execution, assuming its brand has no pending notifications, we must again make a scheduling decision. This time the threads that are candidates for subsequent execution include: (1) the

thread that was previously executing when the upcall occurred, (2) any threads that have been woken up by the upcall's execution, and (3) any additional upcalls that occurred in the meantime (possibly due to interrupts), that were not immediately executed. At the time of this scheduling decision, one option is to upcall into the root scheduler, notifying it that the event completed. It is then possible for other schedulers in the hierarchy to be invoked. Unfortunately, invoking schedulers adds overhead to the upcall, and increases the response time for event notification. We, therefore, propose a novel technique in which the schedulers interact with the kernel to provide hints, using *event structures*, about how to perform subsequent scheduling decisions without requiring their invocation during upcall execution. This technique requires each scheduler in the system to share a private region with the kernel. This region is established by passing the `COS_SCHED_SHARED_REGION` flag to the `cos_sched_cntl` system call detailed in Table 3.1.

Corresponding threads under the control of a given scheduler are then associated with an event structure using the `COS_SCHED_THD_EVT` flag. Each of these event structures has an *urgency* field, used for priority-based scheduling. Depending on the policy of a given scheduler, urgencies can be dynamic (to reflect changing time criticality of a thread, as in the case of a deadline) or static (to reflect different degrees of importance). Numerically lower values for the urgency field represent higher priorities relative to other threads. Within the event structure, there is also a flag section to notify schedulers about the execution status of the thread associated with the event. This is relevant for inactive upcalls, as they are not currently schedulable. The last field of the event structure is an index pointer used to maintain a linked list of pending events that have not yet been recorded by the scheduler.

Event structures are placed in a corresponding shared memory region, accessible from the kernel regardless of which protection domain is currently active. Thus, when an upcall is performed, the event structures for the closest common scheduler in the hierarchy to the currently running thread and the upcall thread are efficiently located, and the urgency values in these structures are compared. If the upcall's brand has a lower numeric urgency field than the current thread, the upcall is immediately executed. The scenario is more

complicated for the case when the upcall is completed. In this case, the kernel checks to see if a scheduler with permissions to schedule the previously executing thread has changed its preference for which thread to run. If this happens it will be reflected via the shared memory region between the scheduler and the kernel. Changes to the scheduling order might be due to the fact that the upcall invoked a scheduler to wake up a previously blocked thread. Additionally the kernel considers if another upcall was made while the current upcall was executing, but is deferred execution. If either of these conditions are true, then an upcall is made into the root scheduler allowing it to make a precise scheduling decision. However, the system is designed around the premise that neither of these cases occur frequently, and most often needs only to switch immediately back to the previous thread. This is typically the case with short running upcalls. However, if the upcalls execute for a more significant amount of time and the root scheduler is invoked, the consequent scheduling overhead is amortized.

Given these mechanisms which allow user-level component schedulers to communicate with the kernel, COMPOSITE supports low asynchronous event response times while still maintaining the configurability of scheduling policies at user-level.

3.1.3 Thread Accountability

Previous research has addressed the problem of accurately accounting for interrupt execution costs and identifying the corresponding thread or process associated with such interrupts [DB96, ZW06]. This is an important factor in real-time and embedded systems, where the execution time of interrupts needs to be factored into task execution times. COMPOSITE provides accurate accounting of the costs of asynchronous event notifications and charges them, accordingly, to corresponding threads.

As stated earlier, brands and upcalls enable efficient asynchronous notifications to be used by the system to deliver events, e.g. interrupts, without the overhead of explicit invocation of user-level schedulers. However, because thread switches can happen without direct scheduler execution, it is more difficult for the schedulers themselves to track total

execution time of upcalls. If the problem were not correctly addressed, then the execution of upcalls might be charged to whatever thread was running when the upcall was initiated. We, therefore, expand the event structure within the shared kernel/scheduler region to include a counter, measuring progress of that event structure's associated thread. In our prototype implementation on the x86 architecture, we use the time-stamp counter to measure the amount of time each thread spends executing by taking a reading whenever threads are switched. The previous reading is subtracted from the current value, to produce the elapsed execution time of the thread being switched out. This value is added to the progress counter in that thread's event structure for each of its schedulers. On architectures without efficient access to a cycle counter, execution time can be sampled, or a simple count of the number of times upcalls are executed can be reported.

Observe that COMPOSITE provides library routines for common thread and scheduling operations. These ease development as they hide event structure manipulation and automatically update thread accountability information.

3.1.4 Efficient Scheduler Synchronization

When schedulers are implemented in the kernel, it is common to disable interrupts for short amounts of time to ensure that processing in a critical section will not be preempted. This approach has been applied to user-level scheduling in at least one research project [GSB+02]. However, given our design requirements for a system that is both dependable and predictable, this approach is not feasible. Allowing schedulers to disable interrupts could significantly impact response time latencies. Moreover, scheduling policies written by untrusted users may have faulty or malicious behavior, leading to unbounded execution (e.g., infinite loops) if interrupts are disabled. CPU protection needs to be maintained as part of a dependable and predictable system design.

An alternative to disabling interrupts is to provide a user-level API to kernel-provided locks, or semaphores. This approach is both complicated and inefficient, especially in the case of blocking locks and semaphores. As blocking is not a kernel-level operation in COM-

POSITE, and is instead performed at user-level, an upcall would have to be performed. However, it is likely that synchronization would be required around wait queue structures, thus producing a circular dependency between kernel locks and the user scheduler, potentially leading to deadlocks or starvation. Additionally, it is unclear how strategies to avoid priority inversion could be included in such a scheme.

Preemptive non-blocking algorithms also exist, that do not necessarily require kernel invocations. These algorithms include both lock-free and wait-free variants [HH01]. Wait-free algorithms are typically more processor intensive, while lock-free algorithms do not necessarily protect against starvation. However, by judicious use of scheduling, lock-free algorithms have been shown to be suitable in a hard-real-time system [ARJ97]. It has also been reported that in practical systems using lock-free algorithms, synchronization delays are short and bounded [HH01, MHH02].

To provide scheduler synchronization that will maintain low scheduler run-times, we optimize for the common case when there is no contention, such that the critical section is not challenged by an alternative thread. We use lock-free synchronization on a value stored in the shared scheduler region, to identify if a critical section has been entered, and by whom. Should contention occur, the system provides a set of synchronization flags that are passed to the `cos_switch_thread` syscall, to provide a form of wait-free synchronization. In essence, the thread, τ_i waiting to access a shared resource "helps" the thread, τ_j, that currently has exclusive access to that resource, by allowing τ_j to complete its critical section. At this point, τ_j immediately switches back to τ_i. The assumption here is that the most recent thread to attempt entry into the critical section has the highest priority, thus it is valid to immediately switch back to it without invoking a scheduler. This semantic behavior exists in a scheduler library in COMPOSITE, so if it is inappropriate for a given scheduler, it can be trivially overridden. As threads never block when attempting access to critical sections, we avoid having to put blocking semantics into the kernel. The design decision to avoid expensive kernel invocations in the uncontested case is, in many ways, inspired by futexes in Linux [FRK02].

Generally, many of the algorithms for non-blocking synchronization require the use of hardware atomic instructions. Unfortunately, on many processors the overheads of such instructions are significant due to factors such as memory bus locking. We have found that using hardware-provided atomic instructions for many of the common scheduling operations in COMPOSITE often leads to scheduling decisions having significant latencies. For example, both the kernel and user-level schedulers require access to event structures, to update the states of upcalls and accountability information, and to post new events. These event structures are provided on a per-CPU basis, and our design goal is to provide a synchronization solution that does not unnecessarily hinder thread execution on CPUs that are not contending for shared resources. Consequently, we use a mechanism called *restartable atomic sequences* (RASes), that was first proposed by Bershad [BRE92], and involves each component registering a list of desired atomic assembly sections. These assembly sections either run to completion without preemption, or are restarted by ensuring the CPU instruction pointer (i.e., program counter) is returned to the beginning of the section, when they are interrupted.

Essentially, RASes are crafted to resemble atomic instructions such as compare and swap, or other such functions that control access to critical sections. Common operations are provided to components via COMPOSITE library routines [1]. The COMPOSITE system ensures that if a thread is preempted while processing in one of these atomic sections, the instruction pointer is *rolled back* to the beginning of the section, similar to an aborted transaction. Thus, when an interrupt arrives in the system, the instruction pointer of the currently executing thread is inspected and compared with the assembly section locations for its current component. If necessary, the instruction pointer of the interrupted thread is reset to the beginning of the section it was executing. This operation performed at interrupt time and is made efficient by aligning the list of assembly sections on cache lines. We limit the number of atomic sections per-component to 4 to bound processing time. The performance benefit of this technique is covered in Section 3.2.1.

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

```
cos_atomic_cmpxchg:
   movl %eax, %edx
   cmpl (%ebx), %eax
   jne cos_atomic_cmpxchg_end
   movl %ecx, %edx
   movl %ecx, (%ebx)
cos_atomic_cmpxchg_end:
   ret
```

Figure 3.4: Example compare and exchange atomic restartable sequence.

Figure 3.4 demonstrates a simple atomic section that mimics the `cmpxchg` instruction in x86. Libraries in COMPOSITE provide the `cos_cmpxchg(void *memory, long anticipated, long new_val)` function which expects the address in memory we wish to change, the anticipated current contents of that memory address, and the new value we wish to change that memory location to. If the anticipated value matches the value in memory, the memory is set to the new value which is returned, otherwise the anticipated value is returned. The library function calls the atomic section in Figure 3.4 with register `eax` equal to anticipated, `ebx` equal to the memory address, `ecx` equal to the new value, and returns the appropriate value in `edx`.

Observe that RASes do not provide atomicity on multi-processors. To tackle this problem, however, either requires the use of true atomic instructions or the partitioning of data structures across CPUs. Note that in COMPOSITE, scheduling queue and event structures are easily partitioned into CPU-specific sub-structures, so our synchronization techniques are applicable to multi-processor platforms.

3.2 Experimental Evaluation

In this Section, we describe a set of experiments to investigate both the overheads of and abilities of component-based scheduling in COMPOSITE. All experiments are performed on IBM xSeries 305 e-server machines with Pentium IV, 2.4 GHz processors and 904 MB of available RAM. Each computer has a tigon3 gigabit Ethernet card, connected by a switched gigabit network. We use Linux version 2.6.22 as the host operating system with a clock-tick (or *jiffy*) set to 10 milliseconds. COMPOSITE is loaded using the techniques from

Hijack [PW07a], and uses the networking device and timer subsystem of the Linux kernel, overriding all other control flow.

3.2.1 Microbenchmarks

Here we report a variety of microbenchmarks investigating the overheads of the scheduling primitives: (1) Hardware measurements for lower bounds on performance. (2) The performance of Linux primitives, as a comparison case. (3) The performance of Composite operating system primitives. All measurements were averaged over 100000 iterations in each case.

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

Table 3.2: Hardware measurements.

Table 3.2 presents the overheads we obtained by performing a number of hardware operations with a minimum number of assembly instructions specially tailored to the measurement. The overhead of switching between user-level to the kernel and back (as in a system call) is 166 cycles. Performing two of these operations approximately doubled the cost. Switching between two protection domains (page-tables), in conjunction with the two system calls, simulates RPC between components in two address spaces. It is notable that this operation on Pentium 4 processors incurs significant overhead.

Operation	Cost in CPU cycles
Null system call	502
Thread switch in same process	1903
RPC between 2 processes using pipes	15367
Send and return signal to current thread	4377
Uncontended lock/release using Futex	411

Table 3.3: Linux measurements.

Table 3.3 presents specific Linux operations. In the past, the `getpid` system call has

been popular for measuring null system call overhead. However, on modern Linux systems, such a function does not result in kernel execution. To measure system-call overhead, then, we use `gettimeofday(NULL, NULL)`, the fastest system call we found. To measure context switching times, We use the NPTL 2.5 threading library. To measure context switch overhead, we switch from one highest priority thread to the other in the same address space using `sched_yield`. To measure the cost of IPC in Linux (an OS that is not specifically structured for IPC), we passed one byte between two threads in separate address spaces using pipes. To understand how expensive it is to create an asynchronous event in Linux, we generate a signal which a thread sends to itself. The signal handler is empty, and we record how long it takes to return to the flow of control sending the signal. Lastly, we measure the uncontended cost of taking and releasing a `pthread_mutex` which uses Futexes [FRK02]. Futexes avoid invoking the kernel, but use atomic instructions.

Operation	Cost in CPU cycles
RPC between components	1629
Kernel thread switch overhead	529
Thread switch w/ scheduler overhead	688
Thread switch w/ scheduler and accounting overhead	976
Brand made, upcall not immediately executed	391
Brand made, upcall immediately executed	3442
Upcall dispatch latency	1768
Upcall terminates and executes a pending event	804
Upcall immediately executed w/ scheduler invocations	9410
Upcall dispatch latency w/ scheduler invocation	5468
Uncontended scheduler lock/release	26

Table 3.4: COMPOSITE measurements.

A fundamental communication primitive in COMPOSITE is a synchronous invocation between components. Currently, this operation is of comparable efficiency to other systems with a focus on IPC efficiency such as L4 [Sto07]. We believe that optimizing the fast-path in COMPOSITE by writing it in assembly can further reduce latency. Certainly, the performance in COMPOSITE is an order of magnitude faster than RPC in Linux (as shown in Table 3.4).

As scheduling in COMPOSITE is done at user-level to ease customization and increase reliability, it is imperative that the primitive operation of switching between threads is not prohibitive. The kernel overhead of thread switching when accounting information is not recorded by the kernel is 0.22 microseconds. This is the lower bound for scheduling efficiency in COMPOSITE. If an actual fixed-priority scheduler is used to switch between threads which includes manipulating run-queues, taking and releasing the scheduler lock, and parsing event structures, the overhead is increased to 0.28 microseconds. Further, if the kernel maintains accounting information regarding thread run-times, and passes this information to the schedulers, overhead increases to 0.40 microseconds. The actual assembly instruction to read the time-stamp counter (`rdtsc`) contributes 80 cycles to the overhead, while locating and updating event structures provides the rest. We found that enabling kernel accounting made programming user-schedulers significantly easier. Even in this form, the thread switch latency is comparable to user-level threading packages that do not need to invoke the kernel, as reported in previous research [vBCZ+03], and is almost a factor of two faster than in Linux.

The overhead and latency of event notifications in the form of brands and upcalls is important when considering the execution of interrupt triggered events. Here we measure overheads of upcalls made under different conditions. First, when an upcall is attempted, but its urgency is not greater than the current thread, or if there are no inactive upcalls, the overhead is 0.16 microseconds. Second, when an upcall occurs with greater urgency than the current thread, the cost is 1.43 microseconds (assuming the upcall immediately returns). This includes switching threads twice, two user $\rightarrow$ kernel round-trips, and two protection domain switches. The time to begin executing an upcall, which acts as a lower-bound on event dispatch latency, is 0.73 microseconds. This is less than a thread switch in the same process in Linux. Third, when an upcall finishes, and there is a pending event, it immediately executes as a new upcall. This operation takes .33 microseconds.

A feature of COMPOSITE is the avoidance of scheduler invocations before and after every upcall. Calling the scheduler both before and after an upcall (that immediately returns) is

3.92 microseconds. By comparison, avoiding scheduler invocations, using COMPOSITE event structures, reduces the cost to 1.43 microseconds. The dispatch latency of the upcall is 2.27 microseconds when the scheduler is invoked, whereas it reduces to 0.73 microseconds using the shared event structures. It is clear that utilizing shared communication regions between the kernel and the schedulers yields a significant performance improvement.

Lastly, we compare the cost of the synchronization mechanism introduced in Section 3.1.4 against futexes. In COMPOSITE, this operation is barely the cost of two function calls, or 26 cycles, compared to 411 cycles with futexes. An illustration of the importance of this difference is that the cost of switching threads, which includes taking and releasing the scheduler lock, would increase in cost by 42% if futexes were used. The additional cost would rise as more event structures are processed using atomic instructions. As thread switch costs bound the ability of the system to realistically allow user-level scheduling, the cost savings is significant.

3.2.2 Case Study: Predictable Interrupt Scheduling

To demonstrate the configurability of the COMPOSITE scheduling mechanisms, we implement a variety of interrupt management and scheduling schemes, and contrast their behavior. A component graph similar to that shown is Figure 3.1 is used throughout our experiments. In this figure, all shaded components are implemented in the kernel. In the experiments in this section, we use network packet arrivals as our source of interrupts, and demultiplex the interrupts based on packet contents [MRA87]. The demultiplexing operation is performed predictably, with a fixed overhead, by carefully choosing the packet parsing method [PR01].

COMPOSITEis configured with a number of different scheduling hierarchies. The scheduling hierarchies under comparison are shown in Figure 3.5. Italic nodes in the trees are schedulers: *HW* is the hardware scheduler giving priority to interrupts, *FP_RR* is a fixed priority round-robin scheduler, and *DS* is a deferrable server with a given execution time and period. All such configurations include some execution at interrupt time labeled the *level 0*

interrupt handling. This is the interrupt execution that occurs before the upcall executes, and in our specific case involves network driver execution. The children below a scheduler are ordered from top to bottom, from higher to lower priority. Additionally, dotted lines signify dependencies between execution entities in the system. The timer interrupt is not depicted, but the *FP_RR* is dependent on it. Task 3 is simply a CPU-bound background task.

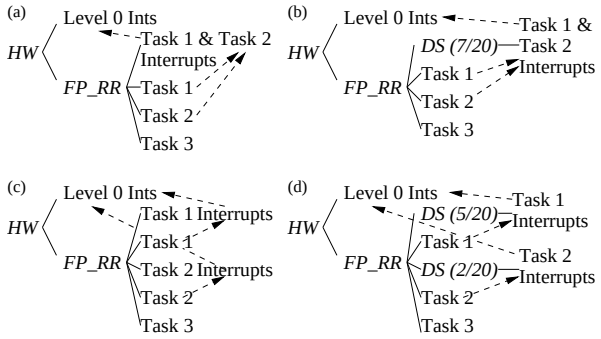

Figure 3.5: Scheduling hierarchies implemented in COMPOSITE.

Figure 3.5(a) depicts a system in which all interrupt handling is executed with the highest priority. Ignoring ad-hoc mechanisms for deferring interrupts given overload (such as `softirqd` in Linux), this hierarchy models the default Linux behavior. Figure 3.5(b) depicts a system whereby the processing of the interrupts is still done at highest priority, but is constrained by a deferrable server [SLS95]. The use of a deferrable server allows for fixed priority schedulability analysis to be performed, but does not differentiate between interrupts destined for different tasks. Figure 3.5(c) depicts a system whereby the interrupts are demultiplexed into threads of different priority depending on the priority of the normal threads that depend on them. This ensures that high priority tasks and their interrupts will be serviced before the lower-priority tasks and their interrupts, encouraging behavior more in line with the fixed priority discipline. The interrupts are processed with higher priority than the tasks, as minimizing interrupt response time is often useful (e.g., to compute accurate TCP round-trip-times, and to ensure that the buffers of the networking card do not overflow, possibly dropping packets for the higher-priority task). Figure 3.5(d) depicts a

system where interrupts for each task are assigned different priorities (and, correspondingly, brands). Each such interrupt is handled in the context of an upcall, scheduled as a deferrable server. These deferrable servers not only allow the system to be analyzed in terms of their schedulability, but also prevent interrupts for the corresponding tasks from causing livelock [MR97].

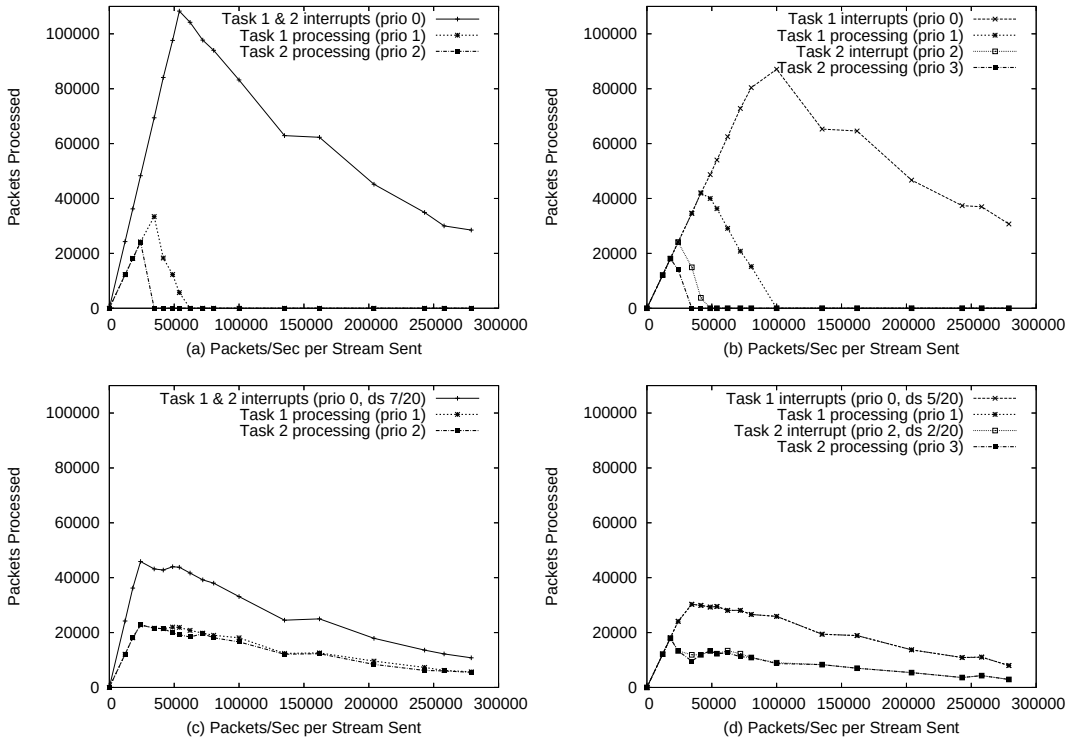

Figure 3.6: Packets processed for two streams and two system tasks.

Streams of packets are sent to a target system from two remote machines, via Gigabit Ethernet. The packets arrive at the host, triggering interrupts, which execute through the device driver, and are then handed off to the COMPOSITE system. Here, a demultiplexing component in the kernel maps the execution to the appropriate upcall thread. From that point on, execution of the interrupt is conducted in a networking component in COMPOSITE, to perform packet processing. This takes 14000 cycles (a value taken from measurements of Linux network bottom halves [FW07]). When this processing has completed, a notification of packet arrival is placed into a mailbox, waking an application task if one is waiting. The

tasks pull packets out of the mailbox queues, and processes them for 30000 cycles.

Figure 3.6(a) depicts the system when the interrupts have the highest priority (NB: lower numerical priority values equate to higher priority, or greater precedence). Packets arriving at sufficient rate cause livelock on the application tasks. The behavior of the system is not consistent or predictable across different interrupt loads. Figure 3.6(b) shows the system configured where the interrupts are branded onto a thread of higher precedence than the corresponding task requesting I/O. In this case, there is more isolation between tasks as the interrupts for Task 2 do not have as much impact on Task 1. Task 1 processes more packets at its peak and performs useful work for longer. Regardless, as the number of received packets increases for each stream, livelock still occurs preventing task and system progress.

Figure 3.6(c) depicts a system that utilizes a deferrable server to execute all interrupts. Here, the deferrable server is chosen to receive 7 out of 20 quanta. These numbers are derived from the relative costs of interrupt to task processing, leading to a situation in which the system is marginally overloaded. An analysis of a system with real-time or QoS constraints could derive appropriate rate-limits in a comparable fashion. In this graph, the interrupts for both tasks share the same deferrable server and packet queue. Given that half of the packets in the queue are from each task, it follows that even though the system wishes one task to have preference (Task 1), they both process equal amounts of packets. Though there is no notion of differentiated service based on Task priorities, the system is able to avoid livelock, and thus process packets across a wide variety of packet arrival rates.

Figure 3.6(d) differentiates interrupts executing for the different tasks by their priority, and also processes the interrupts on two separate deferrable servers. This enables interrupts to be handled in a fixed priority framework, in a manner that bounds their interference rate on other tasks. Here, the high priority task consistently processes more packets than the lower-priority task, and the deferrable servers guarantee that the tasks are isolated from livelock. The cumulative packets processed for Task 1 and 2, and the total of both are plotted in Figure 3.7(c). Both approaches that prevent livelock, by using deferrable servers, maintain high packets processing throughput. However, the differentiated service approach

is the only one that both achieves high throughput, and predictable packet processing (with a 5 to 2 ratio for Tasks 1 and 2).

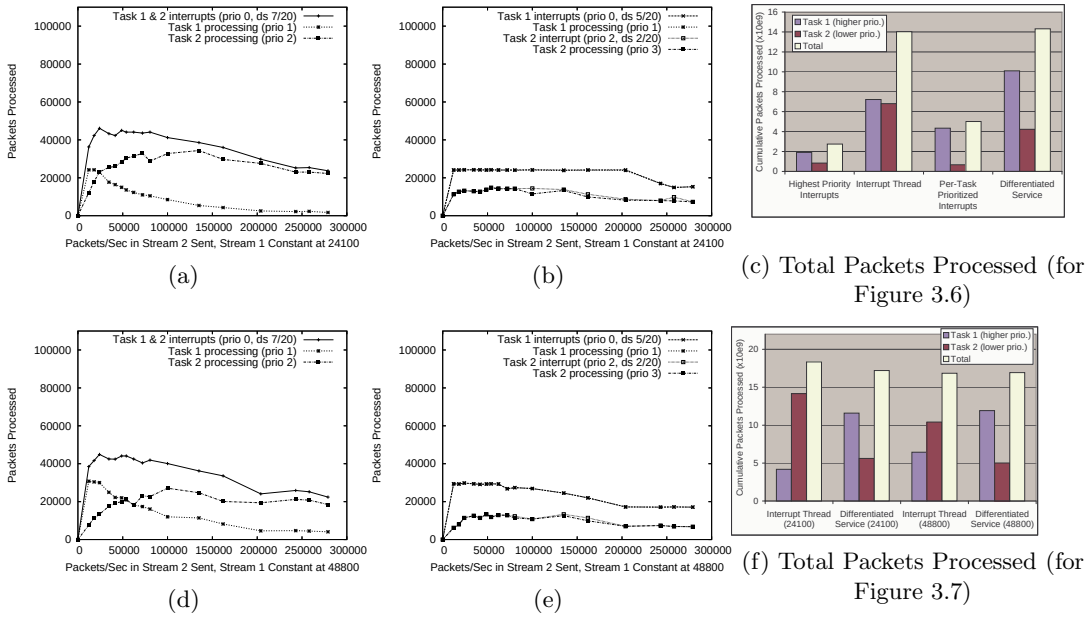

Figure 3.7: Packets processed for two streams, one with constant rate.

Figure 3.7 further investigates the behaviors of the two approaches using deferrable servers. Specifically, we wish to study the ability of the system to maintain predictably differentiated service between the two tasks, given varying interrupt arrival rates. In this case, Task 1 is sent a constant stream of 24100 packets per second in Figure 3.7(a) and (b), and 488000 in Figure 3.7(d) and (e). The amount of packets per second sent to Task 2 varies along the x-axis. The results for both receive rates demonstrate that when all interrupts share the same deferrable server, allocation of processed packets to tasks is mainly dependent on the ratio of packets sent to Task 1 and Task 2. Separating interrupt processing into two different deferrable servers, on the other hand, enables the system to differentiate service between tasks, according to QoS requirements.

Figure 3.7(f) plots the total amounts of packets processed for the tasks in the system under the different hierarchies and constant packet receive rates. The differentiated service approach maintains a predictable allocation of processing time to the tasks consistent with

their relative deferrable server settings. Using only a single deferrable server and therefore ignoring the task dependencies on interrupts, yields processing time allocations that are heavily skewed towards the task of lesser importance when it has more packets arrivals.

3.3 Conclusions

This chapter presents the design of user-level, component-based schedulers in the COMPOSITE component-based system. Separating user-level scheduling policies from the kernel prevents them from compromising the underlying system's integrity. Moreover, component services themselves are isolated from one another, thereby avoiding potentially adverse interactions. Collectively, this arrangement serves to provide a system framework that is both extensible and dependable. However, to ensure sufficient predictability for use in real-time domains, COMPOSITE features a series of low-overhead mechanisms, having bounded costs that are on par or better than competing systems such as Linux. Microbenchmarks show that COMPOSITE incurs low overhead in its various mechanisms to communicate between and schedule component services.

We describe a novel method of branding upcall execution to higher-level thread contexts. We also discuss the COMPOSITE approach to avoid direct scheduler invocation while still allowing full user-level control of scheduling decisions. Additionally, a lightweight technique to implement non-blocking synchronization at user-level, essential for the manipulation of scheduling queues, is also described. This is similar to futexes but does not require atomic instructions, instead relying on "restartable atomic sequences".

We demonstrate the effectiveness of these techniques by implementing different scheduling hierarchies, featuring various alternative policies, and show that it is possible to implement differentiated service guarantees. Experiments show that by using separate deferrable servers to handle and account for interrupts, a system is able to behave according to specific service constraints, without suffering livelock.

Chapter 4

Mutable Protection Domains: Policy

Fault isolation on modern systems is typically limited to coarse-grained entities, such as segments that separate user-space from kernel-level, and processes that encapsulate system and application functionality. For example, systems such as Linux simply separate user-space from a monolithic kernel address space. Micro-kernels provide finer-grained isolation between higher-level system services, often at the cost of increased communication overheads. Common to all these system designs is a static structure, that is inflexible to changes in the granularity at which fault isolation is applicable.

For the purposes of ensuring behavioral correctness of a complex software system, it is desirable to provide fault isolation techniques at the smallest granularity possible, while still ensuring predictable software execution. For example, while it may be desirable to assign the functional components of various system services to separate protection domains, the communication costs may be prohibitive in a real-time setting. That is, the costs of marshaling and unmarshaling message exchanges between component services, the scheduling and dispatching of separate address spaces and the impacts on cache hierarchies (amongst other overheads) may be unacceptable in situations where deadlines must be met. Conversely, multiple component services mapped to a single protection domain experience minimal communication overheads but lose the benefits of isolation from one another.

This chapter focuses on this trade-off between component communication costs, and the

fault-isolation benefit they provide. We formulate a model for a component-based system, and investigate algorithms that make the trade-off between isolation and performance by maximizing the number of protection boundaries while still meeting application performance constraints such as deadlines. Using such an algorithm in a MPD policy component enables the dynamic restructuring of a component-based system, to maximize isolation utility while maintaining timeliness.

The rest of the chapter is organized as follows. Section 4.1 provides an overview of the system, and formally defines the problem being addressed. System dynamics and proposed solutions to the problem are then described. An experimental evaluation is covered in Section 4.2, and finally, conclusions are discussed in Section 4.3.

4.1 Component-Based System Model

With mutable protection domains, isolation between components is increased when there is a resource surplus, and is decreased when there is a resource shortage. Such a system, comprising fine-grained components or services, can be described by a directed acyclic graph (DAG), where each node in the graph is a component, and each edge represents inter-component communication (with the direction of the edge representing control flow). Represented in this fashion, a functional hierarchy becomes explicit in the system construction. Multiple application tasks can be represented as subsets of the system graph, that rely on lower-level components to manage system resources. Component-based systems enable system and application construction via composition and have many benefits, specifically to embedded systems. They allow application-specific system construction, encourage code reuse, and facilitate quick development. Tasks within the system are defined by execution paths through a set of components.

A natural challenge in component-based systems is to define where protection domains should be placed. Ideally, the system should maximize component isolation (thereby increasing system dependability in a beneficial manner) while meeting constraints on application

tasks. Task constraints can vary from throughput goals to memory usage, to predictable execution within a worst-case execution time (WCET). A task's WCET is formulated assuming a minimal amount of fault isolation present within the system. A schedule is then constructed assuming this WCET, and the implicit requirement placed on the system is that the task must complete execution within its allotted CPU share. In most cases, the actual execution time of tasks is significantly lower than the pessimistic worst case. This surplus processing time within a task's CPU allocation can be used to increase the isolation between components (inter-component communication can use the surplus CPU time).

In general, task constraints on a system with mutable protection domains can be defined in terms of multiple different resources. We focus on timeliness constraints of tasks in this paper, and consider only a single resource (i.e., CPU time) for communication between, and execution of, components. For n tasks in a system, each task, τ_k, has a corresponding target resource requirement, RT_k, which is proportional to its worst-case execution time. The measured resource usage, RM_k, is the amount of τ_k's resource share utilized by its computation. Similarly, the resource surplus for τ_k is RS_k, where $RS_k = RT_k - RM_k$. For n tasks the resource surplus is represented as a vector, $\vec{RS} = \langle RS_1, \ldots, RS_n \rangle$.

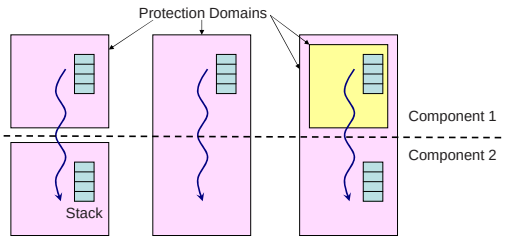

Figure 4.1: Example Isolation Levels.

In response to resource surpluses, different levels of isolation can be placed at the boundaries between components, depending upon their inter-component communication overheads. Three possible *isolation levels* are depicted in Figure 4.1. On the left, we see complete hardware isolation, equivalent to process-level protection in UNIX-type systems, which incurs costs in terms of context switching between protection domains. In the center, we have no isolation, equivalent to how libraries are linked into the address space of code that

uses them. Such a lack of isolation implies only function-call overheads for inter-component communication. Finally, the right-hand side of Figure 4.1 depicts an asymmetric form of isolation, whereby component 1 is *inside* the protection domain of component 2 but not vice versa. This isolation scheme is equivalent to that found in many monolithic OSes such as Linux, which separate the kernel from user-space but not vice versa. It is also similar to the scheme used in our User-Level Sandboxing approach [WP06].

Problem Definition: By adapting isolation levels, which in turn affects inter-component communication costs, we attempt to increase the robustness of a software system while maintaining its timely execution. The problem, then, is to find a system configuration that maximizes the benefit of fault isolation, while respecting task execution (and, hence, resource) constraints. Using a DAG to represent component interactions, let $E = \{e_1, \ldots, e_m\}$ be the set of edges within the system, such that each edge, $e_i \in E$, defines an isolation boundary, or *instance*, between component pairs. For each edge, e_i, there are N_i possible isolation levels, where $N_{max} = max_{\forall e_i \in E}(N_i)$. Where isolation is required and immutable for security reasons, there might exist only one available isolation level ($N_i = 1$), so that security is never compromised. Isolation level j for isolation instance, e_i, is denoted e_{ij}. The overhead, or resource cost, of e_{ij} is $\vec{c_{ij}}$, where $c_{ijk} \in \vec{c_{ij}}, \forall RS_k \in \vec{RS}$. Conversely, each isolation level provides a certain benefit to the system, b_{ij}. We assume that the costs are always lower for lower isolation levels and that the benefit is always higher for higher isolation levels. Finally, $\vec{s}$ denotes a solution vector, where isolation level $s_i \in \{1, \cdots, N_i\}$ is chosen for isolation instance e_i. The solution vector defines a system isolation configuration (or system configuration for short).

More formally, the problem of finding an optimal system configuration is as follows:

$$\begin{aligned} \text{maximize } & \sum_{i \leq m} b_{is_i} \\ \text{subject to } & \sum_{i \leq m} c_{is_ik} \ \leq RS_k, \ \forall RS_k \in \vec{RS} \\ & s_i \in \{1, \ldots, N_i\}, \ \forall e_i \in E \end{aligned} \tag{4.1}$$

Represented in this manner, we have a multi-dimensional, multiple-choice knapsack

problem (MMKP). Though this problem is NP-Hard, approximations exist [Kha98, LLS+99, AMSK01, PHD05]. Specifically, heuristics proposed in these papers attempt to solve the objective function:

$$O(E, \vec{RS}) = \max_{0<j\leq N_i} \{O(E \backslash e_i, \vec{RS} - \vec{c_{ij}}) + b_{ij} \mid e_i \in E\}$$

4.1.1 System Dynamics

Previous approaches to the MMKP for QoS attempt to solve a resource-constrained problem off-line, to maximize system utility. After a solution is found, actual system resources are then allocated. In our case, we wish to alter an existing system configuration *on-line*, in response to resource surpluses or deficits that change over time. The dynamics of the system that introduce changes in resource availability and isolation costs include: (i) threads changing their invocation patterns across specific isolation boundaries, thus changing the overhead of isolation instances throughout the system, (ii) threads altering their computation time within components, using more or less resources, thus changing $\vec{RS}$, and (iii) misprediction in the cost of isolation. Thus, heuristics to calculate system configurations to maximize beneficial isolation over time must adapt to such system dynamics.

It is difficult to compensate for these dynamic effects as the measurements that can be taken directly from the system do not yield complete information. Specifically, at each reconfiguration, the system can measure resource usage, $\vec{RM}$, but explicit information regarding the overhead of isolation is not in the general case observable. For example, the isolation costs between two components mapped to separate protection domains might include context-switch overheads, which in turn have secondary costs on caches, including translation look-aside buffers (TLBs). Such secondary costs are difficult to extract from the total runtime of a task. Section 4.1.6 discusses the impact of mispredicting isolation costs on system behavior.

4.1.2 Dynamic Programming Solution

Given that our problem can be defined as a multi-dimensional, multiple-choice knapsack problem, there are known optimal dynamic programming solutions, For comparison, we describe one such approach similar to that in [LLS+99].

$$DP[i,j,\vec{RS}] = \begin{cases} \max(-\infty, DP[i,j-1,\vec{RS}]) & if \underset{RS_k \in \vec{RS}}{\forall} c_{ijk} > RS_k \\ b_{ij} & if\ i = 1 \\ take(i,1,\vec{RS}) & if\ j = 1 \\ \max(take(i,j,\vec{RS}), DP[i,j-1,\vec{RS}]) & otherwise \end{cases}$$

$$take(i,j,\vec{RS}) \equiv b_{ij} + DP[i-1, N_{i-1}, \vec{RS} - \vec{c_{ij}}]$$

Figure 4.2: Dynamic programming solution.

Figure 4.2 shows the dynamic programming solution DP. The algorithm is initially invoked with $DP[|E|, N_{|E|}, \langle RS_1, \ldots, RS_n \rangle]$. For the lowest isolation level (level 1) of an edge, we assume the sum of the minimal isolation levels is always within the resource consumption limits. That is, $\forall k, \sum_{e_i \in E} c_{i1k} \leq RS_k$.

The recurrence keeps track of the current resource usage and iterates through all isolation levels for a given instance, choosing that which provides the best benefit given its resource cost. The base cases are when we have run out of resources or reached the last isolation instance ($i = 1$).

The complexity of the algorithm reflects the memory structure used: $O(|E| \cdot N_{max} \cdot RS_1 \cdot \ldots \cdot RS_n)$. Because of the memory and execution time requirements, this algorithm is impractical for on-line use, but is useful for comparison.

4.1.3 `HEU`

`HEU` is a heuristic solution first proposed in [Kha98] that is summarized here. `HEU` is a popular comparison case in the MMKP literature and is competitive in terms of quality of solution. Previous algorithms, `HEU` included, assume that the knapsack is initially empty and choose items to place in it from there. This algorithm's general strategy is to weight isolation

benefits versus their costs to choose which isolation level to increase. It uses Toyoda's notion of an aggregate resource cost [Toy75] to collapse multiple constraint dimensions into one by penalizing those dimensions that are more scarce. Then, `HEU` uses a greedy algorithm based on benefit density to choose the isolation level that will yield the highest benefit for the least resource usage. This chosen isolation level is added to the system configuration. This process is repeated, new aggregate resource costs are chosen and that isolation level with the best benefit density is chosen until the resources are expended. Because the aggregate resource cost is recomputed or refined when choosing each edge, we will refer to this algorithm as using *fine-grained refinement*. The asymptotic time complexity of this algorithm is $O(|E|^2 \cdot N_{max}^2 \cdot |\vec{RS}|)$.

4.1.4 Computing Aggregate Resource Costs

Aggregate resource costs should have higher contributions from resources that are scarce, thereby factoring in the cost per unit resource. Inaccurate aggregate costs will lead to a system that does not evenly distribute resource usage across its task constraint dimensions. The approach we take to computing costs is similar to that in [LLS+99]. First, we compute an initial penalty vector which normalizes the total resource costs across all isolation instances and levels, $\vec{c_{ij}}$, by the vector of available resources, $\vec{RS}$. This is shown in Equation 4.2 and will be subsequently referred to as `init_penalty_vect`, $\vec{p}$.

$$\vec{p} = \langle p_1, \cdots, p_n \rangle \mid p_k \in \vec{p} = \frac{\sum_{\forall e_i \in E} \sum_{j \leq N_i} c_{ijk}}{RS_k} \tag{4.2}$$

$$p_k = \frac{(\sum_{\forall e_i \in E} c_{is_ik}) p'_k}{(\sum_{\forall e_i \in E} c_{is_ik}) + RS_k} \mid p_k \in \vec{p}, p'_k \in \vec{p'} \tag{4.3}$$

Equation 4.3 is used to refine the penalty vector, taking into account the success of the previous value, $\vec{p'}$. Recall from Section 4.1 that s_i is the chosen isolation level for isolation instance e_i, while c_{is_ik} is the cost in terms of resource constraint RS_k. We will subsequently refer to the updated penalty vector calculated from Equation 4.3 as `update_penalty_vect`.

Finally, Equation 4.4 defines the aggregate resource cost, c^*_{ij}, using the most recent penalty vector, $\vec{p}$.

$$c^*_{ij} = \sum_{\forall RS_k \in \vec{RS}} (c_{ijk} - c_{is_ik})(p_k) \tag{4.4}$$

4.1.5 Successive State Heuristic

Our approach to solving the multi-dimensional, multiple-choice knapsack problem differs from traditional approaches, in that we adapt a system configuration from the current state. By contrast, past approaches ignore the current state and recompute an entirely new configuration, starting with an empty knapsack. In effect, this is equivalent to solving our problem with a system initially in a state with minimal component isolation.

Our solution, which we term the *successive state heuristic* (`ssh`), successively mutates the current system configuration. `ssh` assumes that the aggregate resource cost, c^*_{ij}, for all isolation levels and all instances has already been computed, as in Section 4.1.4. Edges are initially divided into two sets: set H comprises edges with higher isolation levels than those in use for the corresponding isolation instances in the current system configuration, while set L comprises edges at correspondingly lower isolation levels. Specifically, $e_{ij} \in H$, $\forall j > s_i$ and $e_{ij} \in L$, $\forall j < s_i$. Each of the edges in these sets are sorted with respect to their change in *benefit density*, $(b_{ij} - b_{is_i})/c^*_{ij}$. Edges in L are sorted in increasing order with respect to their change in benefit density, while those in H are sorted in decreasing order. A summary of the `ssh` algorithm follows (see **Algorithm 1** for details):

(i) While there is a deficit of resources, edges are removed from the head of set L to replace the corresponding edges in the current configuration. The procedure stops when enough edges in L have been considered to account for the resource deficit.

(ii) While there is a surplus of resources, each edge in H is considered in turn as a replacement for the corresponding edge in the current configuration. If $e_{ij} \in H$ increases the system benefit density and does not yield a resource deficit, it replaces e_{is_i}, otherwise it is added to a dropped list, D. The procedure stops when an edge is reached that yields

a resource deficit.

(iii) At this point, we have a valid system configuration, but it may be the case that some of the edges in H could lead to higher benefit if isolation were lessened elsewhere. Thus, the algorithm attempts to concurrently add edges from the remaining edges in both H and L. A new configuration is only accepted if it does not produce a resource deficit and heightens system benefit.

(iv) If there is a resource surplus, edges from the dropped list, D, are considered as replacements for the corresponding edges in the current configuration.

The cost of this algorithm is $O(|E| \cdot N_{max} \log(|E| \cdot N_{max}))$, which is bounded by the time to sort edges. The `ssh` algorithm itself is invoked via:

(1) **Algorithm 2**. Here, only an initial penalty vector based on Equation 4.2 is used to derive the aggregate resource cost, c^*_{ij}. The cost of computing the penalty vector and, hence, aggregate resource cost is captured within $O(|\vec{RS}| \cdot |E| \cdot N_{max})$. However, in most practical cases the edge sorting cost of the base `ssh` algorithm dominates the time complexity. We call this algorithm `ssh_oneshot` as the aggregate resource cost is computed only once.

(2) **Algorithm 3**. This is similar to Algorithm 2, but uses Equation 4.3 to continuously refine the aggregate resource cost given deficiencies in its previous value. The refinement in this algorithm is conducted after an entire configuration has been found, thus we say it uses *coarse-grained refinement*. This is in contrast to the fine-grained refinement in Section 4.1.3 that adjusts the aggregate resource cost after each *isolation level* is found. We found that refining the aggregate resource cost more than 10 times, rarely increased the benefit of the solution. This algorithm has the same time complexity as `ssh_oneshot`, but does add a larger constant overhead in practice.

4.1.6 Misprediction of Isolation Overheads

The proposed system can measure the number of component invocations across specific isolation boundaries to estimate communication costs. However, it is difficult to measure the cost of a single invocation. This can be due to many factors including secondary

Algorithm 1: `ssh`: Successive State Heuristic

Input: $\vec{s}$: current isolation levels, $\vec{RS}$: resource surplus

$b^*_{ij} = (b_{ij} - b_{is_i})/c^*_{ij}, \forall i,j$ // benefit density change

// sorted list of lower isolation levels

$L =$ `sort_by_`$b^*(\{\forall e_{ij} | e_i \in E \wedge j < s_i\})$

// sorted list of higher isolation levels

$H =$ `sort_by_`$b^*(\{\forall e_{ij} | e_i \in E \wedge j > s_i\})$

$D = \phi$ // dropped set (initially empty)

while $\exists k, RS_k < 0 \wedge L \neq \phi$ **do** // lower isolation

  $e_{ij} =$ `remove_head`(L)

  if $c^*_{ij} < c^*_{is_i}$ **then**

    $\vec{RS} = \vec{RS} + \vec{c_{is_i}} - \vec{c_{ij}}$

    $s_i = j$

end

$e_{ij} =$ `remove_head`(H)

while $(\nexists k, RS_k + c_{is_ik} - c_{ijk} < 0 \vee b^*_{ij} \leq b^*_{is_i}) \wedge e_{ij}$ **do** // raise isolation greedily

  if $b^*_{ij} > b^*_{is_i}$ **then** // improve benefit?

    $\vec{RS} = \vec{RS} + \vec{c_{is_i}} - \vec{c_{ij}}$

    $s_i = j$

  else $D = D \cup e_{ij}$

  $e_{ij} =$ `remove_head`(H)

end

`replace_head`(e_{ij}, H)

// refine isolation considering both lists

$\vec{s'} = \vec{s}$

while $H \neq \phi \wedge L \neq \phi$ **do**

  repeat // expend resources

    $e_{ij} =$ `remove_head`(H)

    if $b^*_{ij} > b^*_{is'_i}$ **then** // improve benefit?

      $\vec{RS} = \vec{RS} + \vec{c_{is'_i}} - \vec{c_{ij}}$

      $s'_i = j$

    else $D = D \cup e_{ij}$

  until $\exists k, RS_k < 0 \vee H \neq \phi$

  while $\exists k, RS_k < 0 \vee L \neq \phi$ **do** // lower isolation

    $e_{ij} =$ `remove_head`(L)

    if $c^*_{ij} < c^*_{is'_i}$ **then**

      $\vec{RS} = \vec{RS} + \vec{c_{is'_i}} - \vec{c_{ij}}$

      $s'_i = j$

  end

  // found a solution with higher benefit?

  if $\sum_{\forall i} b_{is'_i} > \sum_{\forall i} b_{is_i} \wedge \nexists k, R_k < 0$ **then** $\vec{s} = \vec{s'}$

end

while $D \neq \phi$ **do** // add dropped isolation levels

  $e_{ij} =$ `remove_head`(D)

  if $j > s_i \wedge \nexists k, RS_k + c_{is_ik} - c_{ijk} < 0$ **then**

    $\vec{RS} = \vec{RS} + \vec{c_{is_i}} - \vec{c_{ij}}$

    $s_i = j$

end

return $\vec{s}$

Algorithm 2: `ssh_oneshot`

Input: $\vec{RS}$: resource surplus, $\vec{s}$ solution vector
1 $\vec{p}$ = `init_penalty_vect(`$\vec{RS}$, $\vec{s}$`)`
2 c^*_{ij} = `aggregate_resource(`$\vec{p}$, $\vec{c}_{ij}$`)`, $\forall i, j$
3 $\vec{s}$ = `ssh(`$\vec{s}$, $\vec{RS}$`)`
4 **return** $\vec{s}$

Algorithm 3: `ssh_coarse`

Input: $\vec{RS}$: resource surplus, $\vec{s}$ solution vector
1 $\vec{p}$ = `init_penalty_vect(`$\vec{RS}$, $\vec{s}$`)`
2 $i = 0$
`// refine penalty vector`
3 **while** $i < 10$ **do**
4 c^*_{ij} = `aggregate_resource(`$\vec{p}$, $\vec{c}_{ij}$`)`, $\forall i, j$
5 $\vec{s'}$ = `ssh(`$\vec{s}$, $\vec{RS}$`)`
6 $\vec{p}$ = `update_penalty_vect(`$\vec{RS}$, $\vec{p}$, $\vec{s'}$`)`
7 **if** $\sum_{\forall i} b_{is'_i} > \sum_{\forall i} b_{is_i}$ **then**
8 $\vec{s} = \vec{s'}$ `// found better solution`
9 i++
10 **end**
11 **return** $\vec{s}$

costs of cache misses, which can be significant [UDS+02]. Given that the cost of a single invocation can be mispredicted, it is essential to guarantee such errors do not prevent the system converging on a target resource usage. We assume that the average estimate of isolation costs for each resource constraint, or task, k, across all edges has an error factor of x_k, i.e., $estimate = x_k * actual\_overhead$. Values of $x_k < 1$ lead to heuristics underestimating the isolation overhead, while values of $x_k > 1$ lead to an overestimation of overheads. Consequently, for successive invocations of MMKP algorithms, the resource surplus is mis-factored into the adjustment of resource usage. As an algorithm tries to use all surplus resources to converge upon a target resource value, the measured resource usage at successive steps in time, $RM_k(t)$, will in turn miss the target by a function of x_k. Equation 4.5 defines the recurrence relationship between successive adjustments to the measured resource usage, $RM_k(t)$, at time steps, $t = 0, 1, ...$ When $x_k > 0.5$ for the misprediction factor, the system converges to the target resource usage, RT_k. This recurrence relationship applies to heuristics such as `ssh` that adjust resource usage from the current system configuration.

$$
\begin{aligned}
RM_k(0) &= \text{resource consumption at } t = 0 \\
RM_k(t+1) &= RM_k(t) + x_k^{-1} RS_k(t) \mid RS_k(t) = RT_k - RM_k(t) \\
&= x_k^{-1} RT_k + (1 - x_k^{-1}) RM_k(t) \\
\\
RM_k(t) &= x_k^{-1} RT_k (\textstyle\sum_{i=0}^{t-1} (1 - x_k^{-1})^i) + (1 - x_k^{-1})^t RM_k(0) \\
RM_k(\infty) &= \begin{cases} RT_k & \text{if } x_k > 0.5 \\ \infty & \text{otherwise} \end{cases}
\end{aligned}
\tag{4.5}
$$

For algorithms that do not adapt the current system configuration, they must first calculate an initial resource usage in which there are no isolation costs between components. However, at the time such algorithms are invoked they may only have available information about the resource usage for the current system configuration (i.e., $RM_k(t)$). Using $RM_k(t)$, the resource usage for a configuration with zero isolation costs between components (call it RU_k) must be estimated. RU_k simply represents the resource cost of threads executing within components. Equation 4.6 defines the recurrence relationship between successive adjustments to the measured resource usage, given the need to estimate RU_k. In the equation, $\alpha_k(t)$ represents an estimate of RU_k, which is derived from the measured resource usage in the current configuration, $RM_k(t)$, and an estimate of the total isolation costs at time t (i.e., $x_k(\sum_{\forall i} c_{is_i}) \mid RM_k(t) - RU_k = \sum_{\forall i} c_{is_i}$).

$$
\begin{aligned}
RM_k(0) = &\quad \text{resource consumption at } t = 0 \\
\alpha_k(t) = &\quad RM_k(t) - x_k(RM_k(t) - RU_k) \\
\\
RM_k(t+1) = &\quad RU_k + x_k^{-1}(RT_k - \alpha_k(t)) \\
= &\quad x_k^{-1} RT_k + (1 - x_k^{-1}) RM_k(t)
\end{aligned}
\tag{4.6}
$$

Given that Equation 4.6 and 4.5 reduce to the same solution, heuristics that reconfigure a system based on the current configuration and those that start with no component isolation both converge on a solution when $x_k > 0.5$. Equation 4.7 allows the system to estimate the misprediction factor for total isolation costs. This equation assumes that overheads unrelated to isolation hold constant in the system.

$$\begin{aligned} x_k &= \frac{RM_k(n-1)-RT_k}{RM_k(n-1)-RM_k(n)} \\ &= \frac{RS_k(n-1)}{RS_k(n-1)-RS_k(n)} \end{aligned} \tag{4.7}$$

4.2 Experimental Evaluation

This section describes a series of simulations involving single-threaded tasks on an Intel Core2 quadcore 2.66 Ghz machine with 4GB of RAM. For all the following cases, isolation benefit for each isolation instance is chosen uniformly at random in the range $[0, 255]$ [1] for the highest isolation level, and linearly decreases to 0 for the lowest level. Unless otherwise noted, the results reported are averaged across 25 randomly generated system configurations, with 3 isolation levels ($\forall i, N_i = 3$), and 3 task constraints (i.e., $1 \leq k \leq 3$). With the exception of the results in Figures 4.5 and 4.6, the surplus resource capacity of the knapsack is 50% of the total resource cost of the system with maximum isolation. The total resource cost with maximum isolation is 10000 [2].

4.2.1 MMKP Solution Characteristics

In this section we investigate the characteristics of each of the MMKP heuristics. The dynamic programming solution is used where possible as an optimal baseline. We study both the quality of solution in terms of benefit the system accrues and the amount of run-time each heuristic requires. The efficiency of the heuristics is important as they will be run either periodically to optimize isolation, or on demand to lower the costs of isolation when application constraints are not met. In the first experiment, the system configuration is as follows: $|E| = 50$ and the resource costs for each edge are chosen uniformly at random such that $\forall i, k, c_{iN_ik} \in [0, 6]$. Numerically lower isolation levels have non-increasing random costs, and the lowest isolation level has $\forall i, k, c_{i1k} = 0$.

[1] Isolation benefit has no units but is chosen to represent the relative importance of one isolation level to another in a range of [0..255], in the same way that POSIX allows the relative importance of tasks to be represented by real-time priorities.

[2] Resource costs have no units but since we focus on CPU time in this paper, such costs could represent CPU cycles.

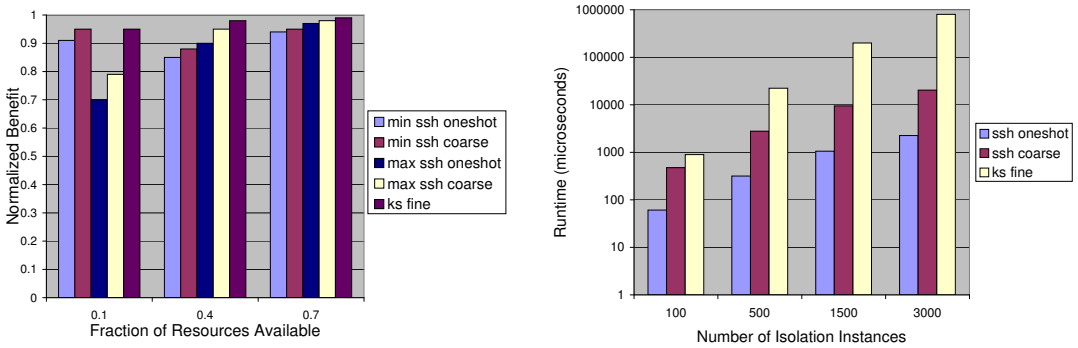

Figure 4.3: MMKP solution characteristics: (a) MMKP benefit, and (b) heuristic run-times.

Figure 4.3(a) shows the normalized benefit of each heuristic with respect to the dynamic programming optimal. The x-axis represents the fraction of the maximal usable resource surplus used as the knapsack capacity. The `ks fine` approach uses the heuristic defined in Section 4.1.3. Heuristics prefixed with `max` start with maximum component isolation, while those prefixed with `min` start with minimal isolation. Generally, the `ks fine` algorithm achieves high benefit regardless of knapsack capacity. The other algorithms achieve a lower percentage of the optimal when they must alter many isolation levels, but the `coarse` refinement versions always achieve higher benefit relative to the `oneshot` approaches. Altering the number of edges does not affect the results significantly, except for very a small number of edges, so we omit those graphs.

Figure 4.3(b) plots the execution times of each heuristic while varying the number of edges in the system. The dynamic programming solution does not scale past 50 edges for 3 task constraints, so is not included here. The `oneshot` algorithms' run-times are dominated by sorting, while all `coarse` algorithms demonstrate a higher constant overhead. Contrarily, the `ks fine` refinement heuristic takes significantly longer to complete because of its higher asymptotic complexity.

4.2.2 System Dynamics

In the following experiments, unless otherwise noted, a system configuration is generated where $|E| = 200$. Resource costs are chosen uniformly at random as follows: $\forall i, k, c_{i3k} \in$

$[0, 100)$, $c_{i2k} \in [0, c_{i3k}]$, and $c_{i1k} = 0$. Note that using resource costs chosen from a bi-modal distribution to model a critical path (i.e., much communication over some isolation boundaries, and little over most others) yield similar results.

Dynamic Communication Patterns: The first dynamic system behavior we investigate is the effect of changing communication patterns between components within the system. Altering the amount of functional invocations across component boundaries affects the resource costs for that isolation instance. Thus, we altered 10% of the isolation instance (i.e., edge) costs after each reconfiguration by assigning a new random cost. All algorithms are able to maintain approximately the same benefit over time. Table 4.1 shows the percentage of isolation instances that have their isolation weakened, averaged over 100 trials.

Algorithm	Isolation Instances with Weakened Isolation
`ks oneshot`	3.4%
`ks coarse`	4.2%
`ssh oneshot`	2%
`ssh coarse`	2.5%
`ks fine`	3%

Table 4.1: Effects of changing communication costs on MPD policies.

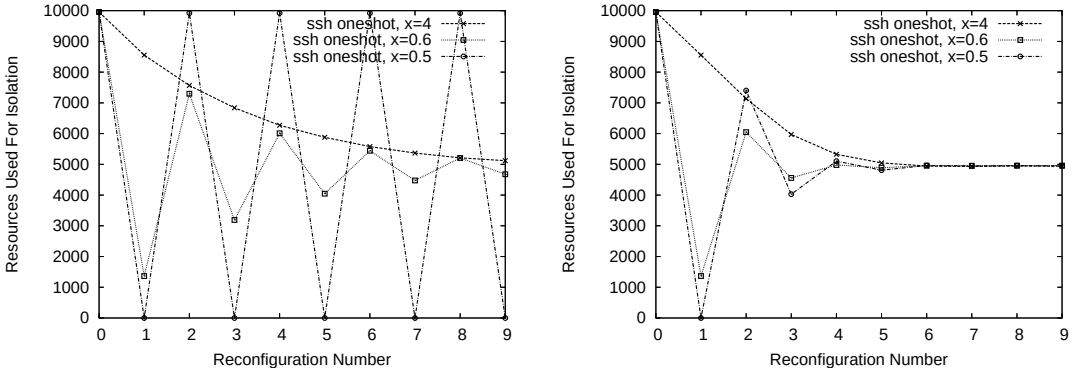

Figure 4.4: Resources used (a) without correction, and (b) with correction for misprediction costs.

Misprediction of Communication Costs: As previously discussed in Section 4.1.6, misprediction of the cost of communication over isolation boundaries can lead to slow convergence on the target resource availability, or even instability. We use the analytical model

in Section 4.1.6 to predict and, hence, correct isolation costs. This is done conservatively, as Equation 4.7 assumes that overheads unrelated to isolation hold constant. However, in a real system, factors such as different execution paths within components cause variability in resource usage. This in turn affects the accuracy of Equation 4.7. Given this, we (1) place more emphasis on predictions made where the difference between the previous and the current resource surpluses is large, to avoid potentially large misprediction estimates due to very small denominators in Equation 4.7, and (2) correct mispredictions by at most a factor of 0.3, to avoid over-compensating for errors. These two actions have the side-effect of slowing the convergence on the target resource usage, but provide stability when there are changes in resource availability.

Figure 4.4(a) shows the resources used for isolation by the `ssh oneshot` policy, when misprediction in isolation costs is considered. Other policies behave similarly. In Figure 4.4(b), the initial misprediction factor, x, is corrected using the techniques discussed previously. The system stabilizes in situations where it does not in Figure 4.4(a). Moreover, stability is reached faster with misprediction corrections than without.

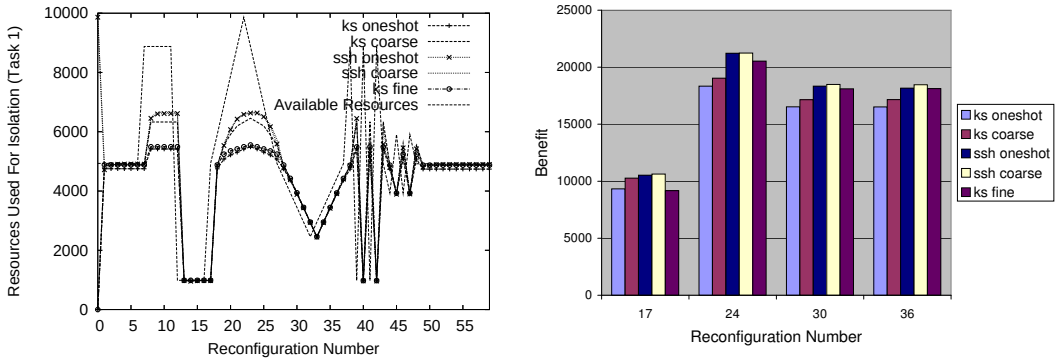

Figure 4.5: Dynamic resource availability: (a) resources consumed by τ_1, and (b) system benefit.

Dynamic Resource Availability: In Figure 4.5(a), the light dotted line denotes a simulated resource availability for task $\tau_k \mid k = 1$. The resources available to τ_2 deviate by half as much as those for τ_1 around the base case of 5000. Finally, resource availability for τ_3 remains constant at 5000. This variability is chosen to stress the aggregate resource cost

computation. Henceforth, traditional knapsack solutions that start with minimal isolation will be denoted by `ks`. Consequently, we introduce the `ks oneshot` and `ks coarse` heuristics that behave as in Algorithms 2 and 3, respectively, but compute a system configuration based on an initially minimal isolation state. We can see from the graph that those algorithms based on `ssh` and `ks coarse` are able to consume more resources than the others, because of a more accurate computation of aggregate resource cost. Importantly, all algorithms adapt to resource pressure predictably. Figure 4.5(b) shows the total benefit that each algorithm achieves. We only plot reconfigurations of interest where there is meager resource availability for τ_1 (in reconfiguration 17), excess resource availability for τ_1 (in 24), a negative change in resource availability (in 30), and a positive change in resource availability (in 36). Generally, those algorithms based on `ssh` yield the highest system benefits, closely followed by `ks fine`.

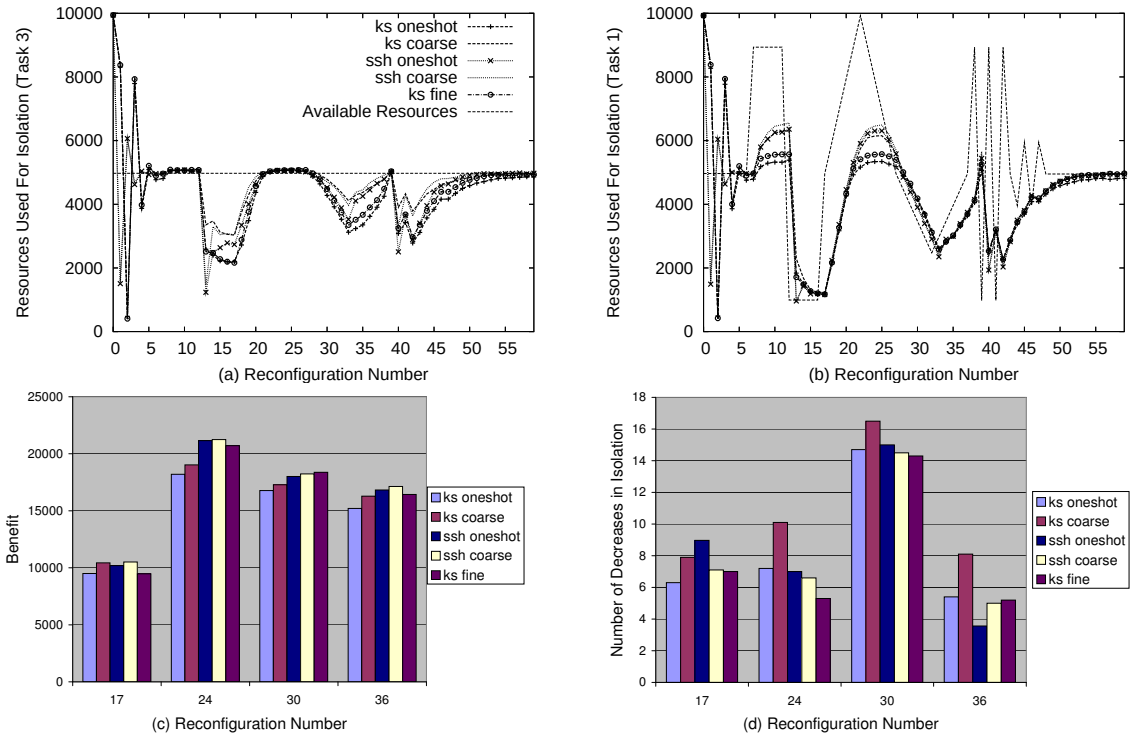

Figure 4.6: Solution characteristics given all system dynamics.

Combining all Dynamic Effects: Having observed the behaviors of the different

algorithms under each individual system dynamic, we now consider the effects of them combined together. Here, we change the cost of 10% of the isolation instances in the system, while the resource availability is changed dynamically in a manner identical to the previous experiment. We assume an initial misprediction factor of $x = 0.6$. Additionally, we employ a conservative policy in which the algorithms only attempt to use 30% of all surplus resources for each reconfiguration.

Figure 4.6(a) presents the resource usage of task τ_3. Resource availability is again denoted with the light dotted line. Figure 4.6(b) presents the resource usage for τ_1. Due to space constraints, we omit τ_2. In both cases, the `ssh` algorithms are able to use the most available resource, followed closely by `ks coarse`. The key point of these graphs is that all heuristics stay within available resource bounds, except in a few instances when the resource usage of the current system configuration briefly lags behind the change in available resources. Figure 4.6(c) plots the system benefit for the different algorithms. As in Figure 4.5(b), we plot only reconfigurations of interest. In most cases, algorithms based on `ssh` perform best, followed by `ks fine`. Of notable interest, the `ssh oneshot` algorithm generally provides comparable benefit to `ssh coarse`, which has an order of magnitude longer run-time. Figure 4.6(d) shows the amount of reconfigurations the different algorithms make, that lessen isolation in the system. Although we only show results for several reconfigurations, `ssh oneshot` performs relatively well considering its lower run-time costs.

Next, the feasibility of mutable protection domains is demonstrated by using resource usage traces for a blob-detection application, which could be used for real-time vision-based tracking. The application, built using the opencv library [Ope] is run 100 times. For each run, the corresponding execution trace is converted to a resource surplus profile normalized over the range used in all prior experiments: [0,10000]. We omit graphical results due to space constraints. 17.75% of components maintain the same fault isolation for all 100 system reconfigurations, while 50% maintain the same isolation for at least 15 consecutive system reconfigurations. This is an important observation, because not all

isolation instances between components need to be changed at every system reconfiguration. On average, 86% of available resources for isolation are used to increase system benefit. Over the 100 application trials, task constraints are met 75% of the time. 97% of the time, resource usage exceeds task constraints by no more than 10% of the maximum available for isolation.

4.3 Conclusions

This chapter investigates a collection of policies to control MPDs. The system is able to adapt the fault isolation between software components, thereby increasing its dependability at the potential cost of increased inter-component communication overheads. Such overheads impact a number of resources, including CPU cycles, thereby affecting the predictability of a system. We show how such a system is represented as a multi-dimensional multiple choice knapsack problem (MMKP). We find that, for a practical system to support the notion of mutable protection domains, it is beneficial to make the fewest possible changes from the current system configuration to ensure resource constraints are being met, while isolation benefit is maximized.

We compare several MMKP approaches, including our own successive state heuristic (`ssh`) algorithms. Due primarily to its lower run-time overheads, the `ssh oneshot` algorithm appears to be the most effective in a dynamic system with changing component invocation patterns, changing computation times within components, and misprediction of isolation costs. The misprediction of isolation costs is, in particular, a novel aspect of this work. In practice, it is difficult to measure precisely the inter-component communication (or isolation) overheads, due to factors such as caching. Using a recurrence relationship that considers misprediction costs, we show how to compensate for errors in estimated overheads, to ensure a system converges to a target resource usage, while maximizing isolation benefit.

The key observation here is that heuristic policies exist to effectively adapt the current system configuration to one with an improved dependability and predictability in response

to dynamic execution characteristics of the system. The next chapter discusses the practical issues and feasibility of implementing MPD in a real system.

Chapter 5

Mutable Protection Domains: Design and Implementation

Whereas Chapter 4 investigates the policy used to place protection domains in a system (implemented in the MPD policy component), this chapter focuses on the design and implementation of Mutable Protection Domains in Composite. MPDs represent a novel abstraction with which to manipulate the trade-off between inter-protection domain invocation costs, and the fault isolation benefits they bring to the system. It is not clear, however, how they can be provided in a CBOS that requires efficient component invocations, and must work on commodity hardware, such as that which uses page-tables to provide protection domains. In this chapter, these questions are answered, and an empirical evaluation of a non-trivial web-server application is conducted to study the benefits of MPD. Section 5.1 discusses how component invocations are conducted in Composite. Section 5.2 details the implementation challenges, approaches, and optimizations for MPD. Section 5.3 gives an overview of the design of a component-based web-server that is used to evaluate MPD in Section 5.4. Section 5.5 concludes the chapter.

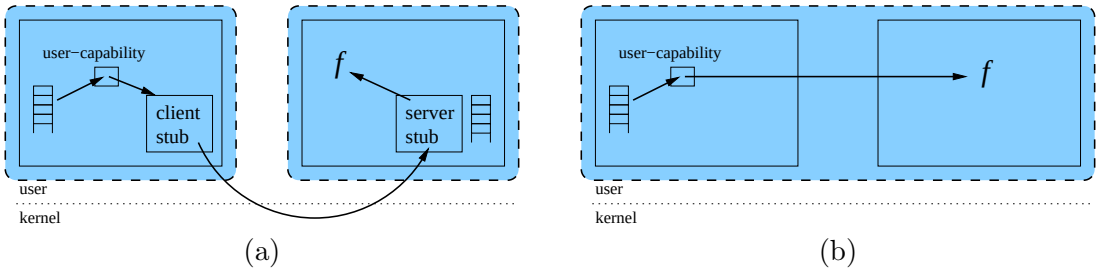

Figure 5.1: Two different invocation methods between components (drawn as the enclosing solid boxes). (a) depicts invocations through the kernel between protection domains (shaded, dashed boxes), (b) depicts intra-protection domain invocations

5.1 Component Invocations

Researchers of μ-kernels have achieved significant advances in Inter-Process Communication (IPC) efficiency [BALL90, Lie95, SSF99, GSB+02]. In additional to the traditional constraint that invocations between components in separate protection domains must incur little overhead, COMPOSITE must also provide intra-protection domain invocations and the ability to dynamically switch between the two modes. This section discusses the mechanisms COMPOSITE employs to satisfy these constraints.

Communication is controlled and limited in COMPOSITE via a capability system [Lev84]. Each function that a component c_0 wishes to invoke in c_1 has an accompanying capability represented by both a kernel- and a user-capability structure. The kernel capability structure links the components, signifies authority allowing c_0 to invoke c_1, and contains the entry instruction for the invocation into c_1. The user-level capability structure is mapped both into c_0 and the kernel. When components are loaded into memory, code is synthesized for each user-capability and linked into c_0. When c_0 invokes the function in c_1, this code is actually invoked and it parses the corresponding user-capability, which includes a function pointer that is jumped to. If intra-protection domain invocations are intended by the MPD system, the function invoked is the actual function in c_1 which is passed arguments directly via pointers. If instead an inter-protection domain invocation is required, a stub is invoked that marshals the invocation's arguments. The current COMPOSITE implementation supports passing up to four arguments in registers, and the rest must be copied. This stub then

invokes the kernel, requesting an invocation on the associated capability. The kernel identifies the component to invoke (c_1), the entry address in c_1 (typically a stub to demarshal arguments), and c_1's page-tables which it loads and, finally, the kernel upcalls into c_1. Each thread maintains an *stack* of capability invocations, and when a component returns from an invocation, the kernel pops off the component to return to. These styles of invocations are depicted in Figure 5.1. To dynamically switch between inter- and intra-protection domain invocations, the kernel need only change the function pointer in the user-level capability structure from a direct pointer to c_1's function to the appropriate stub.

An implication of this component invocation design is that all components that can possibly be dynamically placed into the same protection domain must occupy non-overlapping regions of virtual address space. This arrangement is similar to single address space OSes [CBHLL92] in which all applications share a virtual address space, but are still isolated from each other via protection domains. A number of factors lessen this restriction: (i) those components that will never be placed in the same protection domain (*e.g.* for security reasons) need not share the same virtual address space, (ii) if components grow to the extent that they exhaust the virtual address space, it is possible to relocate them into separate address spaces under the constraint that they cannot be collapsed into the same protection domain in the future, and (iii) where applicable, 64 bit architectures provide an address range that is large enough that sharing it is not prohibitive.

Pebble [GSB+02] uses specially created and optimized executable code to optimize IPC. Composite uses a comparable technique to generate the stub that parses the user-level capability. This code is on the fast-path and is the main intra-protection domain invocation overhead. When loading a component into memory, we generate code that inlines the appropriate user-capability's address to avoid expensive memory accesses. To minimize overhead, we wish to provide functionality that abides by C calling conventions and passes arguments via pointers. To achieve this goal, the specifically generated code neither clobbers any persistent registers nor does it mutate the stack.

The MPD policy component must be able to ascertain where communication bottlenecks

exist. To support this, the invocation path contains counters to track how many invocations have been made on specific capabilities. One count is maintained in the user-level capability structure that is incremented (with an overflow check) for each invocation, and another is maintained in the kernel capability structure. System-calls are provided for the MPD policy component to separately read these counts. In designing invocations in COMPOSITE, we decided to maintain the invocation counter in the user-level capability structure despite the fact that it is directly modifiable by components. When used correctly, the counter provides useful information so that the MPD policy better manipulates the trade-off between fault-isolation and performance. However, if components behave maliciously, there are two cases to consider: (i) components can alter the counter by increasing it to a value much larger than the actual number of invocations made between components which can cause the MPD policy to remove protection domain boundaries, and (ii) components can artificially decrease the counter, encouraging the MPD policy to erect a protection boundary. In the former case, a malicious component already has the ability to make more invocations than would reflect realistic application scenarios, thus the ability to directly alter the counter is not as powerful as it seems. More importantly, a component with malicious intent should never be able to be collapsed into the protection domain of another untrusting component in the first place. In the latter case, components could only use this ability to inhibit their own applications from attaining performance constraints. Additionally, when a protection boundary is erected, the MPD policy will obtain accurate invocation counts from the kernel-capability and will be able to detect the errant user-capability invocation values. Fundamentally, the ability to remove protection domain boundaries is meant to trade-off fault-isolation for performance, and should not be used in situations where a security boundary is required between possibly malicious components. The separation of mechanisms providing fault isolation versus those providing security is not novel [SBL03, LAK09].

As with any system that depends on high-throughput communication between system entities, communication overheads must be minimised. COMPOSITE is unique in that communication mechanisms can switch from inter- to intra-protection domain invocations and

back as the system is running. This process is transparent to components and does not require their interaction. COMPOSITE employs a migrating thread model [FL94] and is able to achieve efficient invocations. Section 5.4.1 investigates the efficiency of this implementation.

5.2 Mutable Protection Domains

A primary goal of COMPOSITE is to provide efficient user-level component-based definition of system policies. It is essential, then, that the kernel provide a general, yet efficient, interface that a MPD policy component uses to control the system's protection domain configuration. This interface includes two major function families: (1) system calls that retrieve information from the kernel concerning the amount of invocations made between pairs of components, and (2) system calls for raising protection barriers, and removing them. In this section, we discuss these in turn.

5.2.1 Monitoring System Performance Bottlenecks

As different system workloads cause diverse patterns of invocations between components, the performance bottlenecks change. It is essential that the policy deciding the current protection domain configuration be informed about the volume of capability invocations between specific pairs of components.

The `cos_cap_cntl(CAP_INVOCATIONS, c0, c1)` system call returns an aggregate of the invocations over all capabilities between component c_0 and c_1, and resets each of these counts to zero. Only the MPD policy component is permitted to execute this call. The typical use of this system call is to retrieve the weight of each edge in the component graph directly before the MPD policy is executed.

5.2.2 Mutable Protection Domain Challenges

Two main challenges in dynamically altering the mapping between components to protection domains are:

(1) How does the dynamic nature of MPD interact with component invocations? Specifically, given the invocation mechanism described in Section 5.1, a thread can be executing in component c_1 on a stack in component c_0, this imposes a lifetime constraint on the protection domain that both c_0 and c_1 are in. Specifically, if a protection boundary is erected between c_0 and c_1, the thread would fault upon execution as it attempts to access the stack in a separate protection domain (in c_0). This situation brings efficient component invocations at odds with MPD.

(2) Can portable hardware mechanisms such as page-tables be efficiently made dynamic? Page-tables consume a significant amount of memory, and creating and modifying them at a high throughput could prove quite expensive. One contribution of COMPOSITE is a design and implementation of MPD using portable hierarchical page-tables that is (1) *transparent* to components executing in the system, and (2) *efficient* in both space and time.

In the rest of Section 5.2 we discuss the primitive abstractions exposed to a MPD policy component used to control the protection domain configuration, and in doing so, reconcile MPD with component invocations and architectural constraints.

5.2.3 Semantics and Implementation of MPD Primitives

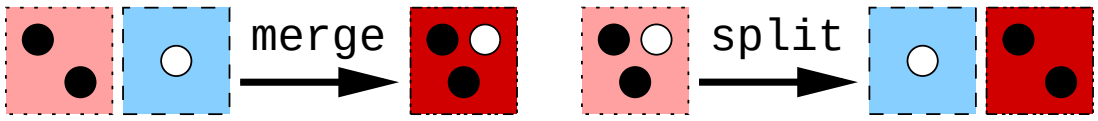

Figure 5.2: MPD `merge` and `split` primitive operations. Protection domain boxes enclose component circles. Different color/style protection domains implies different page-tables.

Two system-calls separately handle the ability to remove and raise protection domain boundaries. `merge(`c_0`, `c_1`)` takes two components in separate protection domains and merges them such that all the components in each co-exist in the new protection domain. This allows the MPD policy component to remove protection domain boundaries, thus communication overheads, between components. A straightforward implementation of these

semantics would include the allocation of a new page-table to represent the merged domain containing a copy of both the previous page-tables. All user and kernel capability data-structures referencing components in the separate protection domains are updated to enable direct invocations, and reference the new protection domain. This operation is depicted in Figure 5.2(a).

To increase the fault isolation properties of the system, the COMPOSITE kernel provides the `split(`$\mathtt{c}_0$`)` system call. `split` removes the specified component from it's protection domain and creates a new protection domain containing only $\mathtt{c}_0$. This ability allows the MPD policy component to improve component fault isolation while also increasing communication overheads. This requires allocating two page-tables, one to contain $\mathtt{c}_0$, and the other to contain all other components in the original protection domain. The appropriate sections of the original page-table must be copied into the new page-tables. All capabilities for invocations between $\mathtt{c}_0$ and the rest of the components in the original protection domain must be updated to reflect that invocations must now be carried out via the kernel (as in Figure 5.1(a)).

Though semantically simple, `merge` and `split` are primitives that are combined to perform more advanced operations. For example, to *move* a component from one protection domain to another, it is split from its first protection domain, and merged into the other. One conspicuous omission is the ability to separate a protection domain containing multiple components into separate protection domains, each with more than one component. Semantically, this is achieved by splitting off one component, thus creating a new protection domain, and then successively *moving* the rest of the components to that protection domain. Though these more complex patterns are achieved through the proposed primitives, there are reasonable concerns involving computational efficiency and memory usage. Allocating and copying page-tables can be quite expensive both computationally and spatially. We investigate optimizations in Section 5.2.5.

5.2.4 Interaction Between Component Invocations and MPD Primitives

Thread invocations between components imply lifetime constraints on protection domains. If in the same protection domain, the memory of the invoking component might be accessed (*i.e.* function arguments or the stack) during the invocation. Thus, if a protection barrier were erected during the call, memory access faults would result that are indistinguishable from erroneous behavior. We consider three solutions to enable the coexistence of intra-protection domain invocations and MPD:

(1) For all invocations (even those between components in the same protection domain), arguments are marshalled and passed via message-passing instead of directly via function pointers, and stacks are switched. This has the benefit of requiring no additional kernel support, but significantly degrades invocation performance (relative to a direct function call with arguments passed as pointers).

(2) For each invocation, a record of the stack base and extent and each argument's base and extent are recorded and tracked via the kernel. In the case that faults due to protection domain and invocation inconsistencies result, these records are consulted, and the relevant pages are dynamically mapped into the current component's protection domain. This option again complicates the invocation path requiring memory allocation for the argument meta-data, and increased kernel complexity to track the stacks and arguments. In contrast to the previous approach, this overhead is not dependent on the size of arguments. A challenge is that once memory regions corresponding to these arguments or the stack are faulted in from an invoking component, how does the kernel track them, and when is it appropriate to unmap them?

(3) The MPD primitives are implemented in a manner that tracks not only the current configuration of protection domains, but also maintains stale protection domains that correspond to the lifetime requirements of thread invocations. This approach adds no overhead to component invocation, but requires significantly more intelligent kernel primitives.

A fundamental design goal of Composite is to encourage the decomposition of the system into fine-grained components on the scale of individual system abstractions and

policies. As OS architects are justifiably concerned with efficiency, it's important that component invocation overheads are removed by the system when necessary. Thus the overhead of intra-protection domain component invocations should be on the order of a C function call. In maintaining a focus on this design goal, COMPOSITE uses the third approach, and investigates if an implementation of intelligent MPD primitives is possible and efficient on commodity hardware using page-tables.

The semantics of the MPD primitives satisfy the following constraint: *All components accessible at the beginning of a thread's invocation to a protection domain must remain accessible to that thread until the invocation returns.* Taking this into account, COMPOSITE explicitly tracks the lifetime of thread's access to protection domains using reference counting based garbage collection. When a thread τ enters a protection domain A, a reference to A is taken, and when τ returns, the reference is released. If there are no other references to A, it is freed. The current configuration of protection domains all maintain a reference to prevent deallocation. In this way, the lifetime of protection domains accommodates thread invocations. The above constraint is satisfied because, even after dynamic changes to the protection domain configuration, *stale* protection domains – those corresponding to the protection domain configuration before a `merge` or `split` – remain active for τ.

There are two edge-cases that must be considered. First, threads might never return from a protection domain, thus maintaining a reference count to it. This is handled by (1) providing system-call that will check if the calling thread is in the initial component invoked when entering into the protection domain, and if so the protection domain mapping for that thread is updated to the current configuration, decrementing the count for the previous domain, and (2) by checking on each interrupt if the above condition is true of the preempted thread, and again updating it to the current protection domain configuration. The first of these options is useful for system-level threads that are aware of MPD's existence and make use of the system-call, and the second is useful for all threads.

The second exceptional case results from a thread's current mappings being out of sync with the current configuration: A thread executing in component c_0 invokes c_1; the current

configuration deems that invocation to be direct as c_0 and c_1 are in the same protection domain; however, if the thread is executing in a stale mapping that doesn't include c_1, a fault will occur upon invocation. In this case, the page-fault path is able to ascertain that a capability invocation is occurring, which capability is being invoked, and if c_0 and c_1 are the valid communicating parties for that capability. If these conditions are true, the stale configuration for the thread is updated to include the mappings for c_1, and the thread is resumed.

Discussion: Efficient intra-protection domain invocations place lifetime constraints on protection domains. In COMPOSITE we implement MPD primitive operations in a manner that differentiates between the current protection domain configuration and *stale* domains that satisfy these lifetime constraints. Protection domain changes take place transparently to components and intra-protection domain invocations maintain high performance: Section 5.4.1 reveals their overhead to be on the order of a C++ virtual function call. When a thread invokes a component in a separate protection domain, the most up-to-date protection domain configuration for that component is always used, and that configuration will persist at least until the thread returns.

5.2.5 MPD Optimizations

The MPD primitives are used to remove performance bottlenecks in the system. If this action is required to meet critical task deadlines in an embedded system, it must be performed in a bounded and short amount of time. Additionally, in systems that switch workloads and communication bottlenecks often, the MPD primitives might be invoked frequently. Though intended to trade-off performance and fault isolation, if these primitives are not efficient, they could adversely effect system throughput.

As formulated in Section 5.2.3, the implementation of `merge` and `split` are not practical. Each operation allocates new page-tables and copies subsections of them. Fortunately, only the page-tables (not the data) are copied, but this can still result in the allocation and copying of large amounts of memory. Specifically, page-tables on ia32 consist of up to 4MB

of memory. In a normal kernel, the resource management and performance implications of this allocation and copying is detrimental. For simplicity and efficiency reasons, the COMPOSITE kernel is non-preemptible. Allocating and copying complete page-tables in COMPOSITE, then, is not practical. This problem is exacerbated by 64 bit architectures with larger page-table hierarchies. Clearly, there is motivation for the OS to consider a more careful interaction between MPD and hardware page-table representations.

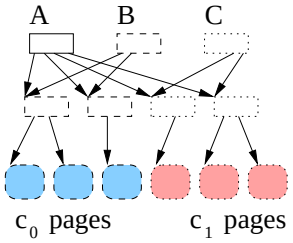

Figure 5.3: COMPOSITE page-table optimization. The top two levels are page-tables, and the shaded bottom level is data pages. Separate protection domains differ only in the top level.

An important optimization in COMPOSITE is that different protection domain configurations do not have completely separate page-tables. Different protection domain configurations differ only in the page table's top level, and the rest of the structure is shared. Figure 5.3 shows three protection domain configurations: an initial configuration, A, and the two resulting from a `split`, B and C. Each different protection domain configuration requires a page of memory, and a 32 byte kernel structure describing the protection domain. Therefore, to construct a new protection domain configuration (via `merge` or `split`) requires allocating and copying only a page.

In addition to sharing second level page-tables which makes MPD practical, COMPOSITE further improves each primitive. In particular, it is important that `merge` is not only efficient, but predictable. As `merge` is used to remove overheads in the system, the MPD policy must be able to mitigate bottlenecks quickly. In real-time systems, it might be necessary to, within a bounded period of time, remove performance overheads so that a critical task meets its deadline. An implication of this is that `merge` must require no memory allocation (the "out-of-memory" case is not bounded in general). To improve

`merge`, we make the simple observation that when merging protection domains A and B to create C, instead of allocating new page-tables for C, A is simply extended to include B's mappings. B's protection domain kernel structure is updated so that its pointer to its page-table points to A's page-table. B's page-table is immediately freed. This places a liveliness constraint on the protection domain garbage collector scheme: A's kernel structure cannot be deallocated (along with its page-tables) until B has no references. With this optimization, `merge` requires no memory allocation (indeed, it frees a page), and requires copying only B's components to the top level of A's page-table.

COMPOSITE optimizes a common case for `split`. A component c_0 is split out of protection domain A to produce B containing c_0 and C containing all other components. In the case where A is not referenced by any threads, protection domain A is reused by simply removing c_0 from it's page-table's top-level. Only B need be allocated and populated with c_0. This is a relatively common case because when a protection domain containing many components is split into two, both also with many components, successive `split`s and `merge`s are performed. As these repeated operations produce new protection domains (*i.e.* without threads active in them), the optimization is used. In these cases, `split` requires the allocation of only a single protection domain and copying only a single component.

The effect of these optimizations is significant, and their result can be seen in Sections 5.4.1 and 5.4.4.

5.2.6 Mutable Protection Domain Policy

This focus of this paper is on the design and implementation of MPD in the COMPOSITE component-based system. However, for completeness, in this section we describe the policy that decides given communication patterns in the system, where protection domain boundaries should exist.

InChapter 4, we introduce a policy for solving for a protection domain configuration given invocations between components and simulate its effects on the system. We adapt that policy to use the proposed primitives. A main conclusion ofChapter 4 is that adapting

the current configuration to compensate for changes in invocation patterns is more effective than constructing a new configuration from scratch each time the policy is executed. The policy targets a threshold for the maximum number of inter-protection domain invocations over a window of time. Thus, the main policy takes the following steps:

(1) remove protection domain barriers with the highest overhead until the target threshold for invocations is met,

(2) increase isolation between sets of components with the lowest overhead while remaining under the threshold, and

(3) refine the solution by removing the most expensive isolation boundaries while simultaneously erecting the boundaries with the least overhead.

It is necessary to understand how the protection boundaries with the most overhead and with the least overhead are found. The policy in this chapter uses a min-cut algorithm [SW97] to find the separation between components in the same protection domain with the least overhead. An overlay graph on the component graph tracks edges between protection domains and aggregates component-to-component invocations to track the overhead of communication between protection domains. These two metrics are tracked in separate priority queues. When the policy wishes to remove invocation overheads, the most expensive inter-protection domain edge is chosen, and when the policy wishes to construct isolation boundaries, the min-cut at the head of the queue is chosen.

5.3 Application Study: Web-Server

To investigate the behavior and performance of MPD in a realistic setting, we present a component-based implementation of a web-server that serves both static and dynamic content (*i.e.* CGI programs) and supports normal HTTP 1.0 connections in which one content request is sent per TCP connection, and HTTP 1.1 persistent connections where multiple requests are be pipelined through one connection. The components that functionally compose to provide these services are represented in Figure 5.4. Each node is a compo-

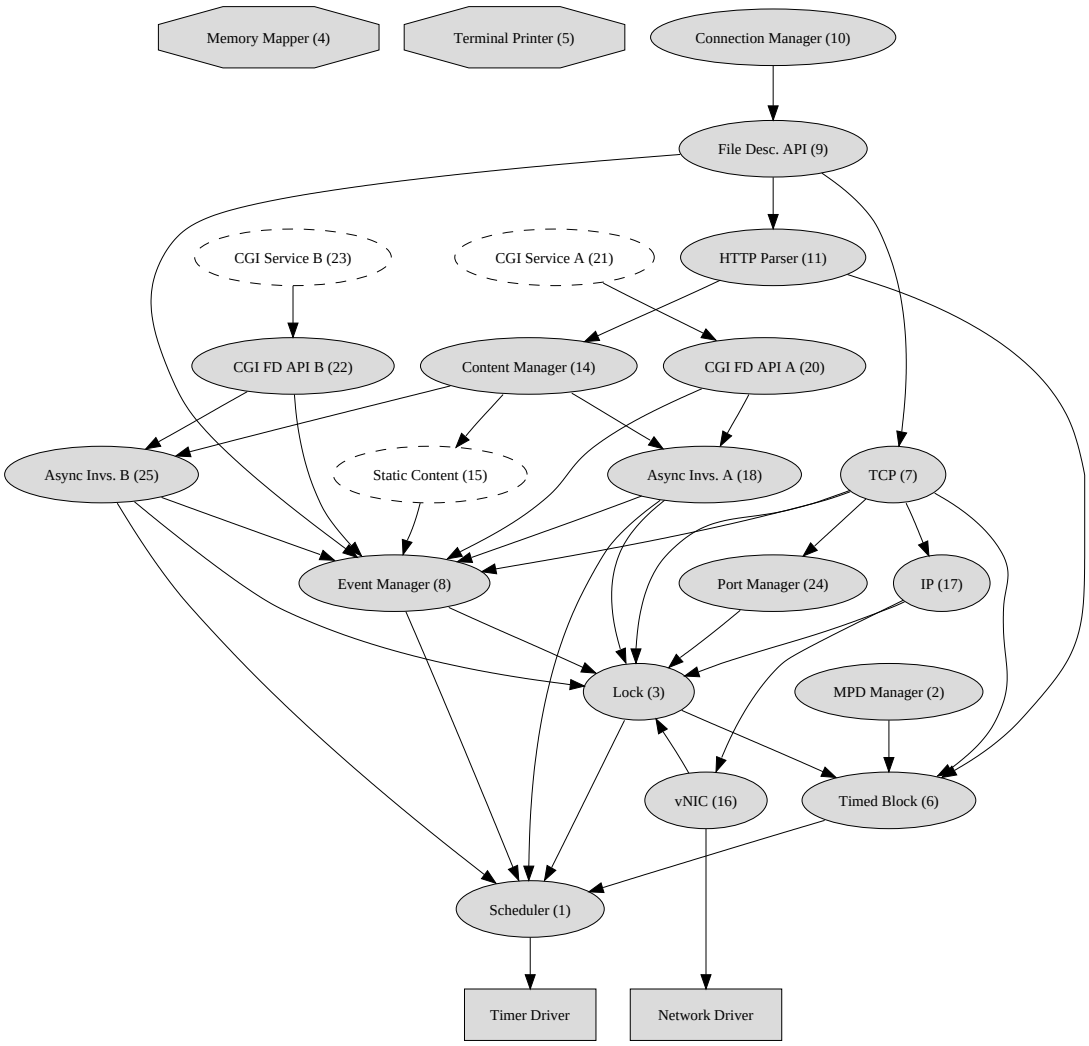

Figure 5.4: A component-based web-server in COMPOSITE

nent, and edges between nodes represent a communication capability. Each name has a corresponding numerical id that will be used to abbreviate that component in some of the results. Rectangular nodes are implemented in the kernel and are not treated as components by COMPOSITE. Nodes that are octogons are relied on for their functionality by all other components thus we omit the edges in the diagram for the sake of simplicity. Indeed, all components must request memory from the *Memory Mapper* component, and, for debugging and reporting purposes, all components output strings to the terminal by invoking the *Terminal Printer*. Nodes with dashed lines represent a component that in a real system

would be a significantly larger collection of components, but are simplified into one for the purposes of this paper. For example, the *Static Content* component provides the content for any non-CGI requests and would normally include at least a buffer cache, a file system, and interaction with a disk device. Additionally, CGI programs are arbitrarily complicated, perhaps communicating via the network with another tier of application servers, or accessing a database. We implement only those components that demonstrate the behavior of a web-server. Note that we represent in the component graph two different CGI programs, A and B. Here too, the component graph could be much more complex as there could be an arbitrarily large number of different CGI programs.

A web-server is an interesting application with which to investigate the effectiveness of MPD, as it is not immediately obvious that MPD is beneficial for it. Systems that exhibit only a single behavior and performance bottleneck, such as many simple embedded systems, wouldn't receive much benefit from dynamic reconfiguration of protection domains. In such systems, MPD could be used to determine an acceptable trade-off between performance and dependability, and that configuration could be used statically (as the performance characteristics are also static). Systems in which workloads and the execution paths they exercise vary greatly over time, such as multi-VM, or possibly desktop systems, could benefit greatly from MPD. As completely disjoint bottlenecks change, so too will the protection domain configuration. A web-server lies somewhere in between. The function of the application is well-defined, and it isn't clear that different bottlenecks will present themselves, thus find benefit in the dynamic reconfiguration of protection domains.

5.3.1 Web-Server Components

We briefly describe how the web server is decomposed into components.

Thread Management:

Scheduler: COMPOSITE has no in-kernel scheduler (as discussed in Chapter 4), instead relying on scheduling policy being defined in a component at user-level. This specific component implements a fixed priority round robin scheduling policy.

Timed Block: Provide the ability for a thread to block for a variable amount of time. Used to provide timeouts and periodic thread wakeups (*e.g.* MPD policy computation, TCP timers).

Lock: Provide a mutex abstraction for mutual exclusion. A synchronization library loaded into client components implements the fast-path of no contention in a manner similar to futexes [FRK02]. Only upon contention is the lock component invoked.

Event Manager: Provide edge-triggered notification of system events in a manner similar to [BMD99]. Block threads that wait for events when there are none. Producer components trigger events.

Networking Support:

vNIC: COMPOSITE provides an virtual NIC abstraction which is used to transmit and receive packets from the networking driver. The *vNIC* component interfaces with this abstraction and provides simple functions to send packets and receive them into a ring buffer.

TCP: A port of lwIP [lwI]. This component provides both TCP and IP.

IP: The TCP component already provides IP functionality via lwIP. To simulate the component overheads of a system in which TCP and IP were separated, this component simply passes through packet transmissions and receptions.

Port Manager: Maintain the port namespace for the transport layer. The TCP component requests an unused port when a connection is created, and relinquishes it when the connection ends.

Web Server Application:

HTTP Parser: Receive a data stream and parse it into separate HTTP requests. Invoke the *Content Manager* with the requests, and when a reply is available, add the necessary headers and return the message.

Content Manager: Receive content requests and demultiplex them to the appropriate content generator (*i.e.* static content, or the appropriate CGI script).

Static Content: Return content associated with a pathname (*e.g.* in a filesystem). As

noted earlier, this component could represent a much larger component graph.

Async. Invocation: Provide a facility for making asynchronous invocations between separate threads in different components. Similar to a UNIX pipe, but bi-directional and strictly request/response based. This allows CGI components to be scheduled separately from the main web-server thread.

File Descriptor API: Provide a translation layer between a single file descriptor namespace to specific resources such as TCP connections, or HTTP streams.

Connection Manager: Ensure that there is a one-to-one correspondence between network file descriptors and *application* descriptors, or, in this case, streams of HTTP data.

CGI Program:

CGI Service: As mentioned before, this component represents a graph of components specific to the functionality of a dynamic content request. It communicates via the *File Descriptor API* and *Async. Invocations* component to receive content requests, and replies along the same channel. These CGI services are persistent between requests and are thus comparable to standard FastCGI [Fas] web-server extensions.

Assorted Others:

The *Memory Mapper* has the capability to map physical pages into other component's protection domains, thus additionally controlling memory allocation. The *Terminal Printer* enables strings to be printed to the terminal. Not shown are the *Stack Trace* and *Statistics Gatherer* components that mainly aid in debugging.

5.3.2 Web-Server Data-Flow and Thread Interactions

As it is important to understand not only each component's functions, but also how they interact, here we discuss the flow of data through components, and then how different threads interact. Content requests arrive from the NIC in the vNIC component. They are passed up through the *IP*, *TCP*, *File Descriptor API* components to the *Connection Manager*. The request is written to a corresponding file descriptor associated with a HTTP session through the *HTTP Parser*, *Content Manager*, and (assuming the request is for

dynamic content) *Async. Invocation* components. The request is read through another file descriptor layer by the *CGI Service*. This flow of data is reversed to send the reply from the *CGI Service* onto the wire.

A combination of three threads orchestrate this data movement. A network thread traverses the *TCP*, *IP*, and *vNIC* components and is responsible for receiving packets, and conducting TCP processing on them. The data is buffered in accordance with TCP policies in *TCP*. The networking thread coordinates with the main application thread via the *Event Manager* component. The networking thread triggers events when data is received, while the application thread waits for events and is woken when one is triggered. Each CGI service has its own thread so as to decouple the scheduling of the application and CGI threads. The application and CGI threads coordinate through the *Async. Invocation* component which buffers requests and responses. This component again uses the *Event Manager* to trigger and wait for the appropriate events.

5.4 Experimental Results

All experiments are performed on IBM xSeries 305 e-server machines with Pentium IV, 2.4 GHz processors and 904 MB of available RAM. Each computer has a tigon3 gigabit Ethernet card, connected by a switched gigabit network. We use Linux version 2.6.22 as the host operating system. Composite is loaded using the techniques from Hijack [PW07a], and uses the networking driver and timer subsystem of the Linux kernel, overriding at the hardware level all other control flow.

5.4.1 Microbenchmarks

Table 5.1 presents the overheads of the primitive operations for controlling MPD in Composite. To obtain these measurements, we execute the `merge` operation on 10 components, measuring the execution time, then `split` the components one at a time, again measuring execution time. This is repeated 100 times, and the average execution time for each op-

Operation	**μ-seconds**
`merge`	0.608
`split` **w/ protection domain reuse**	1.357
`split`	2.130
Clone address space in Linux	30.678

Table 5.1: Mutable Protection Domains primitive operations.

eration is reported. In one case, the kernel is configured to allow the `split` optimization allowing the reuse of a protection domain, and in the other this optimization is disabled.

Cloning an address space in Linux is a mature and efficient operation that is optimized by copying only the page-table, and not data pages. Instead data pages are mapped in copy-on-write. We include in the table the cost of replicating a protection domain in Linux measured by executing the `sys_clone` system call passing in the `CLONE_VM` flag and subtracting the cost of the same operation without `CLONE_VM`. `sys_clone` copies or creates new structures for different process resources. As `CLONE_VM` controls if the address space (inc. page-tables) are copied or not, it allows us to focus on the cost of protection domain cloning. The process that is cloned in the experiments contains 198 pages of memory, with most attributed to libraries such as glibc. The efficiency of Composite MPD primitives is mainly due to the design decision to share all but top-level page-table directories across different MPD configurations. This means that only a single page, the directory, must be created when a new configuration is created. Whereas Linux must copy the entire page-table, Composite must only manufacture a single page. This optimization has proven useful in making MPD manipulation costs negligible.

Operation	**μ-seconds**
Inter-PD component invocation	0.675
Hardware overhead of switching between two PDs	0.462
Intra-PD component invocation	0.008
Linux pipe RPC	6.402

Table 5.2: Component communication operations.

The efficiency of communication and coordination between components becomes increasingly important as the granularity of components shrink. In COMPOSITE, the processing cost of invocations between components in separate protection domains must be optimized to avoid judicious removal of protection domain boundaries via MPD. Additionally, the cost of invocations between components in the same protection domain must be as close to that of a functional call as possible. This is essential so that there is no incentive for a developer to produce otherwise coarser components, effectively decreasing the flexibility and customizability of the system.

Table 5.2 includes microbenchmarks of invocation costs between components. Each measurement is the result of the average of 100000 invocations of a null function on a quiescent system, and thus represent warm-cache and optimistic numbers. An invocation between two components in separate protection domains including two transitions from user to kernel-level, two from kernel to user, and two page-table switches consumes 0.675 μ-seconds. A significant faction of this cost is due to hardware overheads. We constructed a software harness enabling switching back and forth between two protection domains (to emulate an invocation or RPC) with inline user-level code addresses, and the addresses of the page-tables clustered on one cache-line to minimize software overheads and observed a processing cost of 0.462 μ-seconds. With 31.555% software overhead in our implementation, there is some room for improvement. Others have had success replacing the general c-based invocation path with a hand-crafted assembly implementation [Lie93], and we believe this approach might have some success here. However, these results show that little undue overhead is placed in the invocation path, thus MPD coexists with a modern and optimized IPC path.

Importantly, these uncontrollable hardware invocation overheads are avoided by merging components into a single protection domain with an overhead of 0.008 μ-seconds, or 20 cycles. This overhead is on the order of a C++ virtual function call [1] which carries an overhead of 18 cycles. We believe this overhead is small enough that there is no incentive

[1] Using `g++` version 4.1.2

for developers to coarsen the granularity of their components due to performance concerns.

We include the costs of a RPC call between two threads in separate address spaces over a Linux pipe. Linux is not a system structured with a primary goal of supporting efficient IPC, so this value should be used for reference and perspective, rather than direct comparison.

5.4.2 Apache Web-Server Comparison

In measuring web-server performance, we use two standard tools: the apache benchmark program (`ab`) [Apa] version 2.3, and `httperf` [htt] version 0.9.0. We compare against Apache [Apa] version 2.2.11 with logging disabled and FastCGI [Fas] provided by `mod_fastcgi` version 2.4.6.

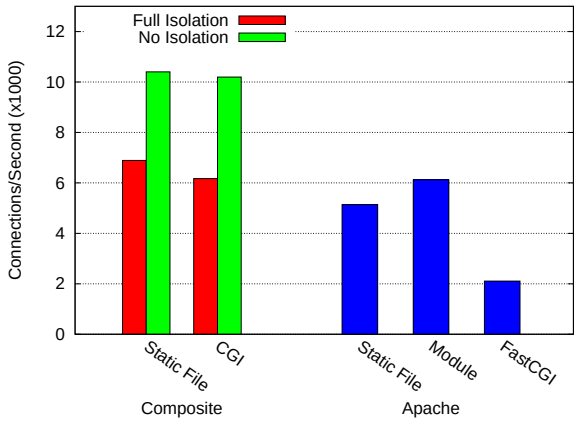

Figure 5.5: Web-server throughput comparison.

Figure 5.5 presents a comparison of sustained connection throughput rates for Composite and different configurations of Apache. Using `ab`, we found that 24 concurrent connections maximizes throughput for all Composite configurations. Requests for static content yield 6891.50 connections/second with a many-to-one mapping of components to protection domains, and 10402.72 connections/second when every component shares the same protection domain. Serving dynamic CGI content yields 6170.74 and 10194.28 connections/second for full and no isolation, respectively. For Apache, we find that 20 concurrent connections

maximizes throughput for serving (cached) static files at 5139.61 connections/seconds, 32 concurrent connections maximizes throughput for module-generated content at 6121.27, and 16 concurrent connections maximizes throughput at 2106.39 connections/second for fastCGI dynamic content. All content sources simply return an 11 character string.

The three Apache configurations demonstrate design points in the trade-off between dependability and performance. Apache modules are compiled libraries loaded directly into the server's protection domain. The module approach co-locates all logic for content generation into the web-server itself. This has the advantage of minimizing communication overhead, but a fault in either effects both. Serving a static file locates the source of the content in the OS filesystem. Accessing this content from the server requires a system-call, increasing overhead, but a failure in the web-server does not disturb that content. Finally, fastCGI is an interface allowing persistent CGI programs to respond to a pipeline of content requests. Because of program persistence, the large costs for normal CGI programs of `fork` and `exec` are avoided. FastCGI, using process protection domains, provides isolation between the dynamic content generator, and the server, but the communication costs are quite high.

The comparison between Composite and Apache is not straightforward. On the one hand, Apache is a much more full-featured web-server than our Composite version which could negatively effect Apache's throughput. On the other hand, Apache is a mature product that has been highly optimized. We propose the most interesting conclusion of these results is validation that a fine-grained component-based system can achieve practical performance levels, and has the ability to increase performance by between 50% and 65% by making some sacrifices in dependability.

5.4.3 The Trade-off Between Fault Isolation and Performance

Next we investigate the trade-off between fault-isolation and performance, and specifically the extent to which isolation must be compromised to achieve significant performance gains. Figure 5.6 presents two separate scenarios in which the server handles (1) static content

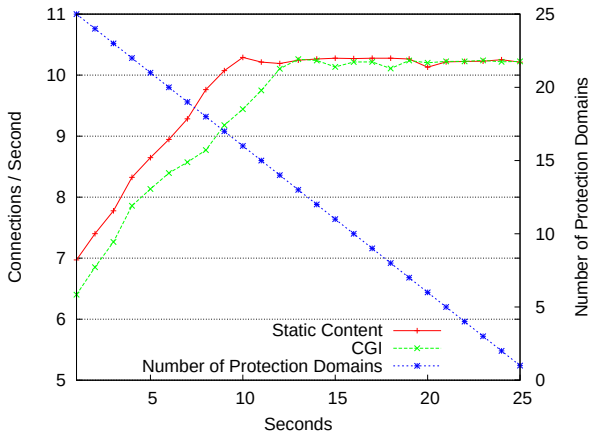

Figure 5.6: The effect on throughput of removing protection domains.

Workload	Sorted Edges w/ > 1% of Total Invocations
Static, HTTP 1.0	10→9, 17→16, 7→17, 9→7, 3→1, 9→11, 9→8, 14→15, 11→14, 7→8, 7→24, 8→1, 7→3, 15→8
CGI, HTTP 1.1	21→20, 20→18, 14→18, 11→14, 18→8, 10→9, 17→16, 7→17, 9→7, 8→1, 3→1, 9→11, 9→8, 7→8

Table 5.3: Edges with the most invocations from Figure 5.7(b) and (c).

requests, and (2) dynamic CGI requests, both generated with `ab` with 24 concurrent requests. The system starts with full isolation, and every second the protection domains with the most invocations between them are merged. When serving static content, by the time 6 protection domains have been merged, throughput exceeds 90% of its maximum. For dynamic content, when 8 protection domains are merged, throughput exceeds 90%. The critical path of components for handling dynamic requests is longer by at least two, which explains why dynamic content throughput lags behind static content processing.

To further understand why removing a minority of the protection domains in the system has a large effect on throughput, Figures 5.7(a) and (b) plot the sorted invocations made over an edge (the bars), and the cumulative distribution function (CDF) of those invocations over a second interval. The majority of the 97 edges between components have zero invocations. Figure 5.7(a) represents the system while processing static requests. Figure 5.7(b) represents the system while processing dynamic requests using HTTP 1.1 persistent con-

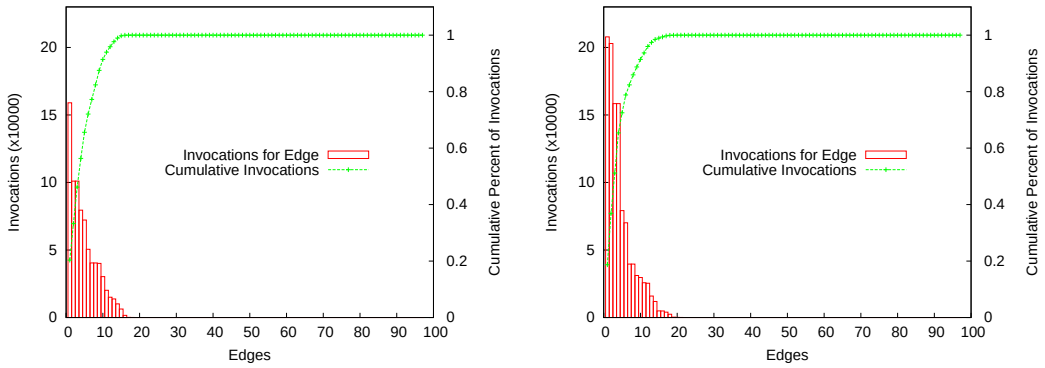

Figure 5.7: The number of invocations over specific threads, and the CDF of these invocations for (a) HTTP 1.0 requests for static content, and (b) persistent HTTP 1.1 requests for CGI-generated content.

nections (generated with `httperf`). In this case, 2000 connections/second each make 20 pipelined `GET` requests. In both figures, the CDF implies that a small minority of edges in the system account for the majority of the overhead. In (b) and (c), the top six edges cumulatively account for 72% and 78%, respectively, of the isolation-induced overhead.

Table 5.3 contains a sorted list of all edges between components with greater than zero invocations. Interestingly, the top six edges for the two workloads contain only a single shared edge, which is the most expensive for static HTTP 1.0 content and the least expensive of the six for dynamic HTTP 1.1 content. It is evident from these results that the bottlenecks for the same system under different workloads differ greatly. It is evident that if the system wishes to maximize throughput while merging the minimum number of protection domains, different workloads can require significantly different protection domain configurations. This is the essence of the argument for dynamic reconfiguration of protection domains.

5.4.4 Protection Domains and Performance across Multiple Workloads

The advantage of MPD is that the fault isolation provided by protection domains is tailored to specific workloads as the performance bottlenecks in the system change over time. To investigate the effectiveness of MPD, Figures 5.8, 5.9, and 5.10 compare two MPD poli-

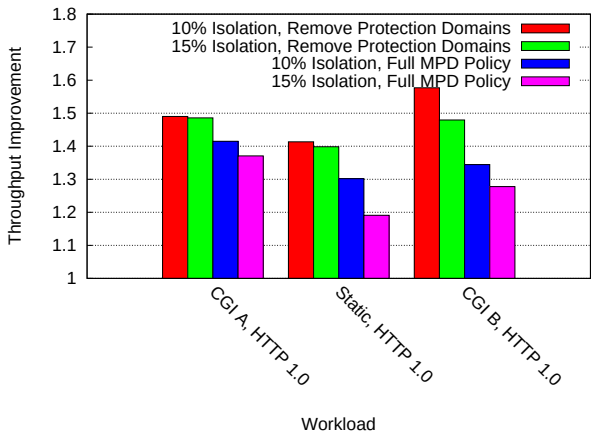

Figure 5.8: The throughput improvement over a system with full isolation.

cies, both of which attempt to keep the number of invocations between protection domains (and thus the isolation overhead) below a threshold. One policy only removes protection boundaries, and the other both merges and splits (labeled in the figure as "Full MPD Policy"). The policy that only merges protection domains represents an oracle of sorts. If the system designer were to statically make the mapping of components to protection domains, the protection domain configuration at the end of the experiments represents a reasonable mapping for them to make. It uses run-time knowledge regarding where invocation bottlenecks exist to adapt the component, protection domain mapping. However, as in the static mapping case, protection domain boundaries cannot be reinstated. We successively execute the system through five different workloads: (1) normal HTTP 1.0 requests for dynamic content from CGI service A, (2) HTTP 1.1 persistent requests (20 per connection) for CGI service B, (3) HTTP 1.0 requests for static content, (4) HTTP 1.1 persistent requests (20 per connection) for CGI service A, (5) HTTP 1.0 requests for dynamic content from CGI service B. A reasonable static mapping of components to protection domains that must address the possibility of all of the workloads would be the protection domain configuration at the conclusion of the experiment. Here we wish to compare MPD with this static mapping.

Each graph includes a plot for two separate policies: one where the threshold for allowed inter-protection domain invocations is set to a calculated 10% of processing time, and the

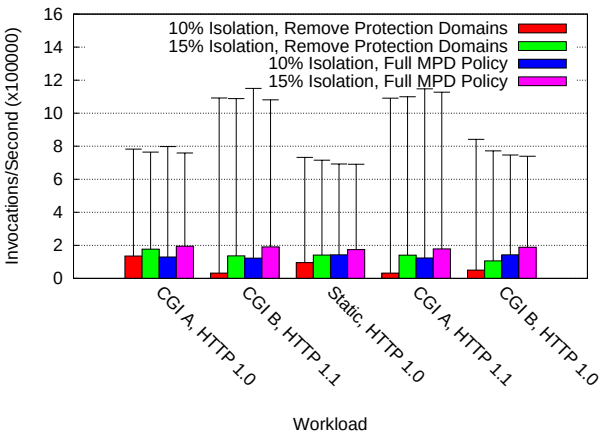

Figure 5.9: The number of inter-protection domain invocations (main bar), and total inter-component invocations (thin bar).

other with it set to 15%. These percentages are calculated by, at system bootup, measuring the cost of a single invocation, and using this to determine how many of such invocations it would take to use the given percent of processing time. 10% corresponds to 142857 invocations, and 15% corresponds to 214285 invocations. These invocation counts approximate the allocated isolation overhead and don't capture cache effects that might change the final overhead. Section 4.1.6 includes a technique for dealing with these unpredictable effects. This is a very simple policy for managing the trade-off between overhead and fault isolation. Certainly more interesting policies taking into account task deadlines, or other application metrics could be devised. However, in this paper we wish to focus on the utility of the MPD mechanisms for reliable systems, and ensure that more complicated MPD policies could be easily deployed as component services.

Figure 5.8 plots the throughput relative to a full isolation system configuration for different workloads. We don't plot the results for the HTTP 1.1 workloads generated with `httperf` as that tool only sends a steady rate of connections/second, instead of trying to saturate the server. All approaches could achieve the sending rate of 2000 connections/second with 20 requests per connection.

All approaches maintain significant increases in performance. It is not surprising that

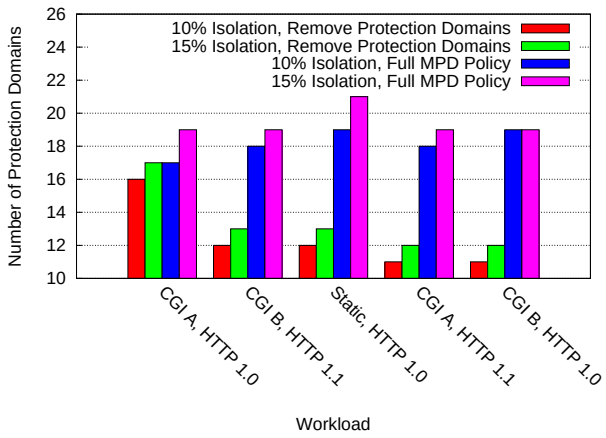

Figure 5.10: The number of active protection domains.

the policies that only remove isolation increase performance over time. The full MPD policies improve performance on average by 35% and 28% for 10% and 15% fault isolation overhead, respectively.

Figure 5.9 measures two factors for each MPD policy, for each workload: (1) the total number of component invocations made (depicted with the line extending above each bar), and (2) the number of invocations that are made between components in separate protection domains. Each policy respects their threshold for the maximum number of inter-protection domain invocations. The number of invocations for the policies that only decreases isolation are significantly below the target thresholds. This indicates that, indeed, there is value in being able to dynamically create protection domain boundaries between components to better use the trade-off between isolation and performance.

This premise is confirmed in Figure 5.10 which plots the number of protection domains (25 being the maximum possible, 1 the minimum). Across all workloads, the policies that both add and remove protection boundaries have on average 18.2 and 19.4 protection domains for isolation overheads of 10% and 15%, respectively. In contrast, the final number of protection domains for the policy that only removes protection domains is 11 and 12 for the two thresholds. This indicates that the full MPD policies are able to adapt to the changing bottlenecks of differing workloads by maintain higher levels of isolation. They do this while

still achieving significant performance gains.

MPD Policy	Component ↔ PD Mapping
	HTTP 1.0 Requests for Static Content
10%, Full MPD	(10,11,9) (17,16,7) (14,15) (3,1)
15%, Full MPD	(17,16,7) (10,11,9)
	HTTP 1.1 Requests to CGI Program A
10%, Full MPD	(21,8,11,14,18,20) (17,16) (10,9)
15%, Full MPD	(11,8,21,20,18,14) (10,9)
	HTTP 1.0 Requests to CGI Program B
10%, Full MPD	(9,10) (16,7,17) (23,25,22) (3,1)
15%, Full MPD	(17,16,7) (9,10) (23,25,22) (3,1)
	After all Workloads
10%, Remove PD Only	(23,21,20,18,10,9,17,16,7,8,11,14,25,22) (3,1)
15%, Remove PD Only	(10,8,7,16,17,9) (11,21,20,18,23,22,25,14) (3,1)

Table 5.4: Protection domain configurations resulting from different workloads and different MPD policies.

Qualitatively, Table 5.4 represents the protection domain configuration for three of the workloads and different MPD policies. Each group of comma-separated components surrounded by parenthesis are co-resident in the same protection domain. Components that are not listed (there are 25 total components) do not share a protection domain. This table demonstrates that it is not only important to observe the number of protection domains in a system, but also how large single protection domains become. By the final workload, the MPD policy that with a tolerance of 10% overhead only removes isolation boundaries has merged all of the most active components into the same protection domain. An error in one of these could trivially propagate to a significant portion of the system. To capture this undesirable behavior, we have found an intuitive *fault exposure* function useful. The fault exposure of a component is defined as the total number of components whereby a fault in them could trivially propagate to this component. Specifically, faults can trivially propagate amongst a single component, but also between components in the same protection domain. Thus the total fault exposure of a system with two components in the same protection domain is 4, as each component is susceptible to its own faults, and to that of the other.

Using this metric, the MPD policy that only removes protection boundaries has a fault exposure averaged across all components of 4.52 and 8.36 for 10% and 15% overhead, respectively, at the end of the experiment. In contrast, the MPD policies that both split and merge protection domains have a fault exposure of 1.864 and 2.376, respectively, averaged across each of the workloads. This reflects that dynamically merging *and* splitting protection domains is effective not only at increasing the number of protection domain boundaries in the system, but also decreasing the possible trivial propagation effects of faults amongst components.

MPD Policy Overhead

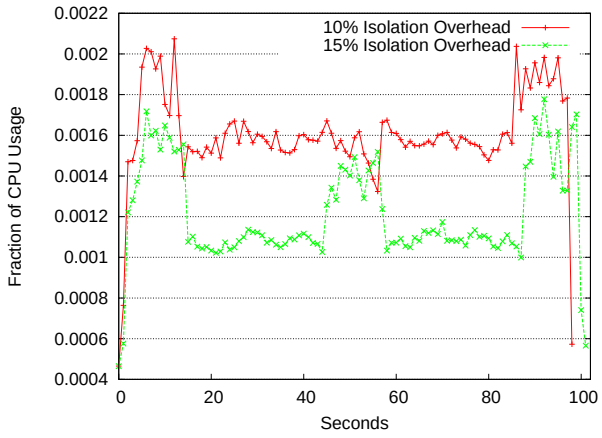

Figure 5.11: Processing overhead of MPD policies.

In addition to the MPD primitive operation's microbenchmarks in Section 5.4.1, here we report the processing costs of executing the MPD policy in a realistic environment. Figure 5.11 depicts the percent of the processing time taken by the thread executing the MPD policy. Each point represents the MPD policy thread's processing time each second. Each second, the MPD policy is run four times. The per-second overhead never exceeds a quarter of a single percent of the total processing time. We conclude that both the primitive operations, and the algorithm for computing the next protection domain configuration are sufficiently efficient to promote frequent adaptation.

5.5 Conclusion

Fine-grained component-based systems are desirable for their software-engineering benefits, and for the system customizability and extensibility they provide. However, due to the trade-off between fault isolation and performance, unless the performance of the system with full isolation is sufficient, a mapping of components to protection domains must be made. The primary hypothesis of this chapter is that a dynamic mapping that adapts to the observed and varying bottlenecks of the system provides a better fault-isolation to performance trade-off than a static mapping. In this chapter, we detail the design and implementation of MPD in the Composite component-based OS including how an efficient IPC system is coordinated with MPD, and an interface used efficiently control MPD. Our proof-of-concept implementation demonstrates how the objective of dynamic protection domains is achieved using commodity and portable architecture features such as hierarchical page tables. Experiments demonstrate that a component-based web-server is competitive in terms of performance with traditionally structured systems, and that MPD be used to achieve a high level of performance while simultaneously maintaining most hardware-provided protection boundaries.

Chapter 6

Future Work

This Chapter describes possible directions in which to take the foundations investigated in this thesis.

6.1 Component-Based Scheduling

Platform for Hierarchical Real-Time Scheduling

In Chapter 3, we discussed how user-level component-based scheduling is provided in the Composite OS. Hierarchical scheduling [RS01] is an important technique for delegating the scheduling policy for different tasks amongst different component schedulers. This technique is useful in, for example, separating tasks with strict timing constraints from best effort tasks in hybrid systems. We provide a prototype of a deferrable scheduler that is hierarchically arranged as a child to the parent fixed priority round robin scheduler.

A generic protocol governing the interaction between child and parent schedulers that does not require the parent to trust the child is required. This protocol will have many of the goals and concerns of scheduler activations [ABLL91] in that child schedulers must inform parent schedulers of idle time, and parent schedulers must communicate information ranging from when tasks block (*e.g.* waiting for I/O) to timer interrupts. The main challenge herein concerns ensuring that the bi-directional communication has (1) a bounded

worst-case execution time for the deliverance of each event, and (2) reasonable average-case overheads. Given a clear communication protocol implemented as a library, this should lower the barrier to empirical hierarchical scheduling research.

Hierarchical Composition of Schedulers

COMPOSITE provides the system mechanisms with which untrusted schedulers are executed as user-level components. The true promise of hierarchical scheduling is that a given set of applications can be coordinated in a manner that respects all their thread's and scheduler's temporal constraints. This thesis provides the mechanisms with which a parent scheduler can orchestrate these child schedulers and threads. An open problem is that of deciding which parent scheduling policy to use and with what parameters, such that the timing constraints of all applications on the system can be met. If all application's requirements aren't mutually satisfiable (*e.g.* they total more than the available resource), and no parent scheduler can provide the required levels of service, a negotiation or admission phase might be required.

Multiprocessor Scheduling

COMPOSITE currently runs on uni-processor platforms. The move to chip multiprocessors requires careful consideration of the necessary scheduling primitives. We anticipate this move in COMPOSITE will revolve around two main difficulties. First, a means for asynchronous, but bounded, communication across cores is necessary. On an architectural level, this is provided by Inter-Processor Interrupts (IPIs), but the abstraction exposed from the COMPOSITE kernel must be more general, not specific to any architecture, and must interoperate with the thread model of the kernel. Second, though restartable atomic sections provide an efficient and convenient mechanism for providing atomic instructions, they do not ensure synchronization across cores. In designing the scheduler data-structures, an investigation of the segregation of the structures into core-local and global sections is required. This decision has implications on where true atomic instructions are required versus being

able to use restartable atomic sections.

6.2 Mutable Protection Domains Policy

The policy determining the mapping from components to protection domains given system communication patterns can be extended in additional directions.

Hierarchical MPD Policy

We found in Chapter 5 that the MPD policy is computationally inexpensive. However, given the super-linear cost of this algorithm, as the system scales up to an order of magnitude more components, the run-time cost might become significant. A possible solution is to decompose the system into multiple *collections* of components. The MPD policy can be run separately on each collection. In such a case, a central MPD policy must coordinate between the different application policies by notifying them how much communication overhead must be removed from their specific collection. If each collection of components is chosen to be the separate applications in the system, the MPD policy is application-specific. This hierarchical policy is useful in that it allows the delegation of MPD policy amongst different application subgraphs of the system. In such a case, each application is allowed to control the placement of protection domains within its components.

MPD for Power Management

As investigated in this thesis, the MPD policy considers CPU performance as the primary resource effected by protection domain placement and the resulting communication overheads. However, the overhead of inter-protection domain communication manifests itself in the utilization of other system resources as well. Specifically, (1) the more protection domains that are active, the more memory they consume (at most a page each) which might be prohibitive on embedded systems, and (2) there is also a correlation between communication overheads and power consumption. The second of these has perhaps greater

implications on MPD policy as voltage or frequency scaling allow discrete changes to the hardware's power consumptions characteristics. The relationship between CPU usage and power savings when considering these scaling techniques is not linear. MPD policy must be adapted to take into account the stepped functions provided by the hardware.

6.3 Composite Design and Implementation

The Composite kernel has been designed and implemented with a number of assumptions including the use of page-tables to provide protection domains and execution on a uni-processor. In improving the capabilities of Composite, these assumptions must be addressed.

Specialized Architectural Support for MPD

Composite's implementation of MPD uses page-tables that are portable across many different architectures. There are some disadvantages of this implementation choice. Namely, this decision requires that switching between different page-tables involves the invalidation of virtually-tagged caches. In commodity x86 processors, this means flushing the Translation Look-aside Buffer (TLB). Reestablishing cache contents induces additional overhead. Some hardware provides Address Space IDs (ASIDs) allowing cache contents to contain an additional token specifying which address space that content is associated with. The use of ASIDs would remove the cost of reestablishing the cache from Composite, but it is not clear how to assign ASIDs to stale protection domains. Other hardware features such as ARM Protection Domains (PDs) can associate different ids with different items in the cache, and allow multiple ids to be active at any point in time. The use of these hardware features has the promise of making inter-protection domain component invocations more efficient in Composite, but additional research must be conducted to understand how they can be made to best interact with MPD. Composite should provide a portable MPD implementation, but should be able to adapt to hardware features when possible to provide

enhanced performance.

MPD on Multiprocessors

The main challenges to implement Composite on multiprocessors are:

(1) maintaining high inter-protection domain invocation performance (which implies avoiding locks and atomic instructions when accessing kernel data-structures),

(2) maintaining efficient intra-protection domain invocations while still keeping accurate invocation counts between components (which implies avoiding sharing the counter's cache-line across cores),

(3) TLB consistency across different cores (as `merge` and `split` modify protection domains, this consistency might be necessary, but could be quite expensive), and

(4) concurrent execution of MPD primitives (`merge` and `split`) on multiple cores requires coordination in modifying protection domain data-structures. Efficient mechanisms for avoiding concurrent modification to data-structures such as protection domains are essential, but providing them without performance impact to the primitives themselves or the invocation path provides a challenge.

We believe each of these problems is addressable, but a thorough investigation is required.

Component Repository

The Composite kernel provides the basis for the development of component-based systems. An interesting direction for future research involves the development of a corpus of components that are practically useful in designing general, yet extensible OSes. In designing the components, emphasis must be placed on the ability to use a wide variety of different implementations throughout the system, thus providing the most configurability for specific applications and problem domains.

Chapter 7

Conclusion

This thesis investigates the design and implementation of the COMPOSITE component-based operating system. In placing a primary focus on defining system policies in user-level components, COMPOSITE enables system extensibility by allowing application-specific components to be used, and heightened dependability by allowing protection domains to be placed around individual components. The fundamental contribution of this thesis is the component-based control of both the temporal- and memory-isolation properties of the system. A key observation is that the fault-isolation between components provided by protection domains does not have to be binary, and can instead adapt to the temporal constraints and execution patterns of the system. Fault-isolation between components can exist at one time and not at another dependent on application inter-protection domain communication overheads, and the temporal requirements of the application. Combined with component-based scheduling, this marks a shift from existing systems by empowering the system to explicitly control the trade-off between fault-isolation and performance, and thus have greater control over the ability of the system to meet application temporal requirements while still maintaining high fault-isolation.

Specific contributions of this thesis follow:

1. The design and implementation of untrusted, user-level component schedulers that can efficiently control both application tasks, and the execution resulting from asyn-

chronous hardware events. This enables the system scheduler to be tailored to a specific application-domain's requirements, and isolates the scheduler from faults within the rest of the system and vice-versa. In doing so, we describe a system implementation that solves how to provide synchronization around data-structures within the scheduler, a mechanism for efficiently scheduling execution due to interrupts, and the coexistence of schedulers with other system mechanisms such as component invocation.

2. In recognizing the trade-off between fault-isolation and communication costs between components, we present Mutable Protection Domains that enable the system to erect or remove protection domain boundaries between components in response to the communication performance bottlenecks. In investigating MPD, we focus on two problems:

   i. Given a graph of components, and the number of invocations made between each component in a unit of time, we determine where to place protection domain boundaries in the system. In doing so, we formulate the problem as a multi-dimensional multiple-choice knapsack problem, and find efficient heuristics for solving it. Importantly, we conclude that algorithms that formulate the next configuration of protection domains from the previous are more effective than those that always start from a no isolation configuration. We show that even in many cases when the system mispredicts the cost of isolation, it will converge on the desired configuration and we provide a framework to estimate and correct for this misprediction.

   ii. We study the feasibility of a design and implementation of MPD in Composite. We introduce simple primitives that are used to control the configuration of protection domains in the system, and detail how they coexist with efficient inter-component communication, and are efficient, both in memory usage and processing cost. Though the MPD mechanism is efficient, the rate at which it can be expected to compute new configurations is limited by its processing time. An open area of investigation

is to determine what is an ideal rate of recomputation of the protection domain configuration. If computed on too small a time-scale, the invocation counts between components will not reflect system-wide communication overheads, and only the computation that took place since the last configuration was chosen. If computed on too large a time-scale, the protection-domain configuration will not adapt to changing workloads and quickly provide beneficial performance improvements.

3. An empirical evaluation of the COMPOSITE system in the context of a non-trivial web-server application. This evaluation demonstrates that for this application a CBOS (with a component-based scheduler) has performance levels competitive with monolithic kernels and applications. Additionally, fault protection boundaries are sacrificed where communication overheads are high to further increase system throughput. We find that in the web-server, the location within a system of the highest communication throughput differs significantly across workloads, and the dynamic nature of MPD is better able than static configurations to trade-off fault-isolation for performance. To further validate the COMPOSITE system, empirical studies into other application domains are necessary.

Bibliography

[ABLL91] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *SOSP '91: Proceedings of the thirteenth ACM Symposium on Operating Systems Principles*, pages 95–109, New York, NY, USA, 1991. ACM Press.

[AMSK01] M. Mostofa Akbar, Eric G. Manning, Gholamali C. Shoja, and Shahadat Khan. Heuristic solutions for the multiple-choice multi-dimension knapsack problem. In *ICCS '01: Proceedings of the International Conference on Computational Science-Part II*, pages 659–668, London, UK, 2001. Springer-Verlag.

[ANSG05] Tejasvi Aswathanarayana, Douglas Niehaus, Venkita Subramonian, and Christopher Gill. Design and performance of configurable endsystem scheduling mechanisms. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 32–43, Washington, DC, USA, 2005. IEEE Computer Society.

[Apa] Apache server project: http://httpd.apache.org/.

[ARJ97] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 15(2):134–165, 1997.

[BALL90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, 1990.

[BM02] Luciano Porto Barreto and Gilles Muller. Bossa: a language-based approach to the design of real-time schedulers. In *10th International Conference on Real-Time Systems (RTS'2002)*, pages 19–31, Paris, France, mar 2002.

[BMD99] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[BRE92] Brian N. Bershad, David D. Redell, and John 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.

[BSP+95] Brian N. Bershad, Stefan Savage, Prsemyslaw Pardyak, Emin Gum Sirer, Marc Fiuczynski, and Becker Eggers Chambers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, December 1995.

[CBHLL92] Jeffrey S. Chase, Miche Baker-Harvey, Henry M. Levy, and Edward D. Lazowska. Opal: A single address space system for 64-bit architectures. *Operating Systems Review*, 26(2):9, 1992.

[CKF+04] 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.

[DB96] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): a network subsystem architecture for server systems. In *OSDI '96: Proceedings of the*

second symposium on Operating Systems Design and Implementation, pages 261–275, New York, NY, USA, 1996. ACM Press.

[Doua] Arnold, douglas, 2000, the explosion of the Ariane 5: http://www.ima.umn.edu/ arnold/disasters/ariane.html.

[Doub] Isbell, douglas, 1999, Mars Climate Orbiter team finds likely cause of loss: http://mars.jpl.nasa.gov/msp98/news/mco990930.html.

[EKO95] Dawson R. Engler, Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 251–266, Copper Mountain Resort, Colorado, USA, December 1995. ACM.

[Ent] Enterprise JavaBeans: http://java.sun.com/products/ejb/index.jsp.

[FAH+06] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, , and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of EuroSys 2006*, pages 177–190, April 2006.

[Fas] FastCGI: http://www.fastcgi.com.

[FHL+96] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Operating Systems Design and Implementation*, pages 137–151, 1996.

[FL94] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the winter 1994 USENIX Technical Conference and Exhibition*, pages 97–114, 1994.

[FRK02] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast user-level locking in Linux. In *Ottawa Linux Symposium*, 2002.

[FS96] Bryan Ford and Sai Susarla. CPU inheritance scheduling. In *OSDI '96: Proceedings of the second symposium on Operating Systems Design and Implementation*, pages 91–105, New York, NY, USA, 1996. ACM Press.

[FSH+01] L. Fernando Friedrich, John Stankovic, Marty Humphrey, Michael Marley, and John Haskins. A survey of configurable, component-based operating systems for embedded applications. *IEEE Micro*, 21(3):54–68, 2001.

[FSLM02] 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.

[FW07] Gerald Fry and Richard West. On the integration of real-time asynchronous event handling mechanisms with existing operating system services. In *Proceedings of the 2007 International Conference on Embedded Systems & Applications*, pages 83–90, June 2007.

[GAGB01] Paolo Gai, Luca Abeni, Massimiliano Giorgi, and Giorgio 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.

[GSB+02] Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. The pebble component-based operating system. In *Proceedings of Usenix Annual Technical Conference*, pages 267–282, June 2002.

[HFB05] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 143–153, New York, NY, USA, 2005. ACM.

[HH01] Michael Hohmuth and Hermann Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 217–230, 2001.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for network sensors. In *ASPLOS '00: Proceedings of Architectural Support for Programming Languages and Operating Systems*, November 2000.

[htt] httperf: http://www.hpl.hp.com/research/linux/httperf/.

[JAU] Joint architecture for unmanned systems: http://www.jauswg.org/.

[KAR+06] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a complete operating system. In *Proceedings of EuroSys 2006*, April 2006.

[KEG+97] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector Briceno, Russel Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Janotti, and Kenneth Mackenzie. Application performance and flexibility on Exokernel systems. In *SOSP '97: Proceedings of the Symposium on Operating Systems Principles*, pages 52–65, 1997.

[Kha98] Md. Shahadatullah Khan. *Quality adaptation in a multisession multimedia system: model, algorithms, and architecture*. PhD thesis, University of Victoria, 1998. Adviser-Kin F. Li and Adviser-Eric G. Manning.

[KLGH07] Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. Camkes: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software*, 80(5):687–699, 2007.

[LAK09] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. Recovery domains: an organizing principle for recoverable operating systems. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 49–60, New York, NY, USA, 2009. ACM.

[LES+97] J. Liedtke, K. Elphinstone, S. Schiinberg, H. Hartig, G. Heiser, N. Islam, and T Jaeger. Achieved ipc performance. In *HOTOS '97: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, page 28, Washington, DC, USA, 1997. IEEE Computer Society.

[Lev84] Hank Levy. Capability-based computer systems, 1984.

[Lie93] Jochen 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.

[Lie95] J. Liedtke. On micro-kernel construction. In *SOSP '95: Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM, December 1995.

[Lin] Linux kernel: www.kernel.org.

[LLS+99] Chen Lee, John Lehoczky, Dan Siewiorek, Ragunathan Rajkumar, and Jeff Hansen. A scalable solution to the multi-resource qos problem. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 315, Washington, DC, USA, 1999. IEEE Computer Society.

[LR04] Jean-Claude Laprie and Brian Randell. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[lwI] lwIP: a lightweight TCP/IP stack: http://www.sics.se/ adam/lwip/index.html.

[MHH02] F. Mehnert, M. Hohmuth, and H. Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *RTSS '02: In Proceedings of the 23rd IEEE Real-Time Systems Symposium*, December 2002.

[MKJK99] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *SOSP '99: Proceedings of the 17th Symposium on Operating Systems Principles*, pages 217–231, 1999.

[MR97] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[MRA87] J. Mogul, R. Rashid, and M. Accetta. The packer filter: an efficient mechanism for user-level network code. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, New York, NY, USA, 1987. ACM.

[MSLM91] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. *SIGOPS Operating Systems Review*, 25(5):110–121, 1991.

[NIS] NIST study – the economic impacts of inadequate infrastructure for software testing: http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[Ope] OpenCV: http://opencvlibrary.sourceforge.net/.

[Ous90] John Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, 1990.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[PHD05] Rafael Parra-Hernandez and Nikitas J. Dimopoulos. A new heuristic for solving the multichoice multidimensional knapsack problem. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 35(5):708–717, 2005.

[Pou] Poulsen, kevin, SecurityFocus, 2004, tracking the blackout bug: http://www.securityfocus.com/news/8412.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Lecture Notes in Computer Science*, 2161:121–144, 2001.

[PW07a] Gabriel Parmer and Richard West. Hijack: Taking control of cots systems for real-time user-level services. In *RTAS '07: Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2007.

[PW07b] Gabriel Parmer and Richard 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.

[QPW04] Xin Qi, Gabriel Parmer, and Richard West. An efficient end-host architecture for cluster communication services. In *in Proceedings of the IEEE International Conference on Cluster Computing (Cluster '04)*, September 2004.

[RD05] John Regehr and Usit Duongsaa. Preventing interrupt overload. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 50–58, New York, NY, USA, 2005. ACM Press.

[RS01] John Regehr and John A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 3–14, London, UK, December 2001.

[Ruo06] Sergio Ruocco. User-level fine-grained adaptive real-time scheduling via temporal reflection. In *Proceedings of 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 246–256, 2006.

[RWG93] T. Anderson R. Wahbe, S. Lucco and S. Graham. Software-based fault isolation. In *SOSP '93: Proceedings of the 14th Symposium on Operating Systems Principles*, Asheville, NC, USA, December 1993.

[SABL04] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2004. USENIX Association.

[SBL03] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 207–222, New York, NY, USA, 2003.

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

[SESS96] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *OSDI '96: Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, 1996.

[SLS95] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computing*, 39(9):1175–1185, 1990.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *SOSP '99: Proceedings of the Symposium on Operating Systems Principles*, pages 170–185, 1999.

[Sto81] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[Sto07] Jan Stoess. Towards effective user-controlled scheduling for microkernel-based systems. *SIGOPS Operating Systems Review*, 41(4):59–68, 2007.

[SW97] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of th ACM*, 44(4):585–591, 1997.

[Toy75] Y. Toyoda. A simplified algorithm for obtaining approximate solution to zero-one programming problems. *Management Science*, 21:1417–1427, 1975.

[UDS+02] Volkmar Uhlig, Uwe Dannowski, Espen Skoglund, Andreas Haeberlen, and Gernot Heiser. Performance of address-space multiplexing on the Pentium. Technical Report 2002-1, University of Karlsruhe, Germany, 2002.

[vBCZ+03] R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[WP06] Richard West and Gabriel Parmer. Application-specific service technologies for commodity operating systems in real-time environments. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–13, 2006.

[ZW06] Yuting Zhang and Richard West. Process-aware interrupt scheduling and accounting. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 191–201, Washington, DC, USA, 2006. IEEE Computer Society.

Curriculum Vitae

Gabriel Ammon Parmer
gabep1@cs.bu.edu

Education

Ph.D. Computer Science, Boston University, (Expected) January 2010.
Towards a Dependable and Predictable Component-Based Operating System for Application-Specific Extensibility
Advisor: Richard West

B.A. Computer Science, Boston University, 2003.

Academic Experience

Research Fellow, Boston University 2006 - *current*
Boston, MA

Design and Implementation of COMPOSITE
Designed and implemented HIJACK and the COMPOSITE component-based system. HIJACK provides a mechanism for the safe interposition of application-specific services on the system-call path, thus allowing system specialization in commodity systems. COMPOSITE enables the construction of an Operating System in an application-specific manner from components. Each component defines policies and abstractions (for example schedulers or synchronization primitives) to manage system resources. These systems focus on providing an execution environment that is both dependable and predictable.

Teaching Fellow, Boston University 2004 – 2006
Boston, MA

CS101 and CS111, 6 semesters, total.
Conducted weekly labs for Introduction to Computers (CS101) and Introduction to Programming (CS111), the first class CS majors take. These required precise communication of fundamental concepts to students with no previous experience. Authored the lab curriculum used by multiple Teaching Fellows in coordination with the topics covered in class.

Professional Experience

Distributed OS Research Intern Summer 2005
Seattle, WA. Cray Inc.

Designed and developed a prototype for the networking subsystem of a single-system image distributed OS. Required Linux kernel development, integration with FUSE (filesystems in user-space), client/server event-based socket communication, and implementation of policies for managing global port namespaces and port status.

Application/Library Programmer Summers 1999 – 2001
Los Alamos, NM. Los Alamos National Labs (LANL)

Implemented an application for visualizing benchmark data from massively parallel programs; modified and increased the functionality of a library used for parallel image compositing.

Honors / Awards

- *Best Paper Award* for co-authored paper at the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2006
- Sole recipient of the CS Department's Annual *Research Excellence Award*, 2007-2008
- Runner-up *Best Teaching Fellow Award*, 2005
- US DoEd Graduate Assistance in Areas of National Need Fellowship, 2006-2008
- *Best Poster Presentation Award* for "On the Design and Implementation of Mutable Protection Domains Towards Reliable Component-Based Systems," CS Dept. Industrial Affiliates Research Day, 2008
- Research Fellowship, Boston University, 2008
- National Science Foundation *Travel Grant* to the Real-Time Systems Symposium (RTSS) 2006
- Undergraduate degree with *Cum Laude* honors

Publications

Refereed Conference Proceedings

Richard West and Gabriel Parmer, *Application-Specific Service Technologies for Commodity Operating Systems in Real-Time Environments*, extended version of RTAS '06 paper, accepted for publication in an upcoming ACM Transactions on Embedded Computing Systems.

Gabriel Parmer and Richard West, *Predictable Interrupt Management and Scheduling in the* COMPOSITE *Component-based System*, in Proceedings of the 29th IEEE Real-Time Systems Symposium (RTSS), December 2008

Gabriel Parmer and Richard West, *Mutable Protection Domains: Towards a Component-based System for Dependable and Predictable Computing*, in Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS), December 2007

Richard West and Gabriel Parmer, *Revisiting the Design of Systems for High-Confidence Embedded and Cyber-Physical Computing Environment*, position paper at the NSF High Confidence Cyber-Physical Systems Workshop, July 2007

Gabriel Parmer, Richard West, and Gerald Fry, *Scalable Overlay Multicast Tree Construction for Media Streaming*, in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), June 2007

Gabriel Parmer and Richard West, HIJACK*: Taking Control of COTS Systems for Real-Time User-Level Services*, in Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), April 2007

Richard West and Gabriel Parmer, *A Software Architecture for Next-Generation Cyber-Physical Systems*, position paper at the NSF Cyber-Physical Systems Workshop, October 2006

Richard West and Gabriel Parmer, *Application-Specific Service Technologies for Commodity Operating Systems in Real-Time Environments*, in Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), April 2006.

- **Best paper award**

Xin Qi, Gabriel Parmer and Richard West, *An Efficient End-host Architecture for Cluster Communication Services*, in Proceedings of the IEEE International Conference on Cluster Computing (Cluster), September 2004.

Gabriel Parmer, Richard West, Xin Qi, Gerald Fry and Yuting Zhang, *An Internet-wide Distributed System for Data-stream Processing*, in Proceedings of the 5th International Conference on Internet Computing (IC), June 2004.

Selected Presentations

Gabriel Parmer, *On the Design and Implementation of Mutable Protection Domains Towards Reliable Component-Based Systems*, Poster presented at Industrial Affiliates Research Day, CS Dept., Boston, MA, March 2008

- **Best poster presentation award**

Gabriel Parmer, *Mutable Protection Domains: Towards a Component-based System for Dependable and Predictable Computing*, Presented at Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS), Tucson, AZ, December 2007

Gabriel Parmer and Richard West, *Hypervisor Support for Component-Based Operating Systems*, invited to the poster session at VMware's VMworld, San Francisco, CA, July 2007

Gabriel Parmer, HIJACK*: Taking Control of COTS Systems for Real-Time User-Level Services*, presented at the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Bellevue, WA, April 2007

Richard West and Gabriel Parmer, HIJACK*: Taking Control of COTS Systems to Enforce Predictable Service Guarantees*, presented at VMware, Cambridge, MA, June 2006

Editorial Services

Reviewer for the following conferences:

- Euromicro Conference on Real-Time Systems (ECRTS) in 2004, 2007, and 2008
- Real-Time and Embedded Technology and Applications Symposium (RTAS) in 2006 and 2007
- Real-Time Systems Symposium (RTSS) in 2004, 2005, 2006, and 2008
- Workshop on Parallel and Distributed Real-Time Systems (WPDRTS) in 2005

References

Research

Richard West - Associate Professor, Boston University
richwest@cs.bu.edu - 1.617.353.2065

Azer Bestavros - Professor, Boston University
best@cs.bu.edu - 1.617.353.9726

Assaf Kfoury - Professor, Boston University
kfoury@cs.bu.edu - 1.617.353.8911

Teaching

Wayne Snyder - Associate Dean for Undergraduate Programs, Boston University
snyder@cs.bu.edu - 1.617.358.2739