Mutable Protection Domains: Adapting System Fault Isolation for Reliability and Efficiency

Gabriel Parmer, *Member*, *IEEE*, and Richard West, *Member*, *IEEE*

Abstract—As software systems are becoming increasingly complex, the likelihood of faults and unexpected behaviors will naturally increase. Today, mobile devices to large-scale servers feature many millions of lines of code. Compile-time checks and offline verification methods are unlikely to capture all system states and control flow interactions of a running system. For this reason, many researchers have developed methods to contain faults at runtime by using software and hardware-based techniques to define protection domains. However, these approaches tend to impose isolation boundaries on software components that are static, and thus remain intact while the system is running. An unfortunate consequence of statically structured protection domains is that they may impose undue overhead on the communication between separate components. This paper proposes a new runtime technique that trades communication cost for fault isolation. We describe *Mutable Protection Domains* (MPDs) in the context of our COMPOSITE operating system. MPD dynamically adapts hardware isolation between interacting software components, depending on observed communication "hot-paths," with the purpose of maximizing fault isolation where possible. In this sense, MPD naturally tends toward a system of maximal component isolation, while collapsing protection domains where costs are prohibitive. By increasing isolation for low-cost interacting components, MPD limits the scope of impact of future unexpected faults. We demonstrate the utility of MPD using a webserver, and identify different hot-paths for different workloads that dictate adaptations to system structure. Experiments show up to 40 percent improvement in throughput compared to a statically organized system, while maintaining high-fault isolation.

Index Terms—Component-based, operating systems, reliability, fault isolation, performance.

1 INTRODUCTION

COMPLEX systems software featuring millions of lines of code is common on widespread platforms ranging from those in mobile computing (e.g., the iPhone) to servers for large-scale data centers. In many cases, the functionality of such software is critical for maintaining high availability, reducing costs [30] and even avoiding potential loss of life. However, software faults are an inevitability of a complex system in which it is impossible to statically verify all possible control flow interactions and memory references. While static analysis techniques have been shown to be useful in helping to detect software bugs [5], [10], they cannot be expected to guarantee reliability at all times.

Many fault-tolerant systems have been developed to isolate and recover from errors at runtime. A fundamental requirement of such systems is the ability to limit the adverse side effects of faults. Many systems are constructed from software components so that the errant behavior of any one component does not adversely impact other parts of the system [8], [28], [25], [38], [12]. Such software composition can increase availability as the system need only recover from the effects of those components that are faulty [35], [11], [37], [23], [6].

Unfortunately, increased fault isolation typically comes at some decrease in system performance. For example, in UNIX systems, Interprocess Communication (IPS) increases both reliability and communication costs compared to function calls within the same address space. Significant effort has gone into limiting the overhead from interprotection domain communication [4], [27], [28], [14], but the costs of these operations on commodity hardware are still relative expensive [31], [27]. In recognition of the tradeoff between fault isolation and performance, some systems [15], [24] support the manual placement of protection boundaries between software components. In such systems, the most expensive communication paths between components can be configured to use fast communication primitives (i.e., function calls) by sacrificing fault isolation. Unfortunately, the system designer is not always able to predict the communication paths that will carry the most overhead due to a number of factors: 1) The applications to be run on the system and, thus, the communication paths that carry the highest cost are not always known a priori, 2) even if the target applications are known, different workloads may result in different communication patterns, and 3) different applications contending for resources may lead to dynamically variable communication costs. System designers are consequently forced to make the tradeoff between performance and fault isolation with incomplete information about the runtime behavior of the system. In such cases, it is common to see designers focus on performance at the expense of reliability [24], [22].

- *G. Parmer is with the Department of Computer Science, The George Washington University, 801 22nd Street NW, Suite 703, Washington, DC 20052. E-mail: gparmer@gwu.edu.*
- *R. West is with the Department of Computer Science, Boston University, 111 Cummington Street, Boston, MA 02215. E-mail: richwest@cs.bu.edu.*

Manuscript received 13 June 2010; revised 10 Jan. 2011; accepted 16 Apr. 2011; published online 16 June 2011.
Recommended for acceptance by H. Schmidt.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-06-0176.
Digital Object Identifier no. 10.1109/TSE.2011.61.

To address the tradeoff between system performance and reliability, this paper proposes a novel mechanism called *Mutable Protection Domains* (MPDs). MPD marks a conceptual shift from a system with fault-isolation boundaries that are fixed at runtime to one that supports *dynamically* placed isolation boundaries around software components. MPD controls *where* protection domains are placed at runtime so that fault isolation is maximized under the constraint that system performance requirements are met. In this way, the configuration of protection domains is tailored to application execution characteristics, and the system controls the tradeoff between reliability and performance.

In this paper, we focus on protection domains that provide memory isolation: Execution cannot access the memory outside of its protection domains, either intentionally or accidentally. Protection domains provide memory isolation by limiting access to the regions of memory. We focus on protection domain support provided by hardware page-tables. This has two motivations: 1) Modern architectures providing hardware memory isolation typically include support for page-tables and thus our techniques should be general across processors, and 2) focusing on memory isolation at the hardware level instead of relying on memory access guarantees provided by a language runtime (e.g., as in the Java Virtual Machine) enables us to provide memory isolation independent of any language runtimes (or lack thereof) executing on the system. Even low-level C or assembly code benefits from this memory isolation.

A fundamental assumption of many systems [28], [25], [19], [40] is that protection domains provided by hardware mechanisms (e.g., page-tables) limit the scope of side effects of errant behavior, and that finer grained fault isolation will result in more dependable systems. This argument is emphasized by significant motivation behind μ-kernels: to break monolithic bodies of software into smaller servers, segregated via memory isolation, to increase the fault isolation of the system. The intuition is that with finer grained protection domains, the memory regions and data structures that faulty code has access to are reduced; thus less of the system can be corrupted when it is brought into an inconsistent state—which we will refer to as a *failure*.

The failure model we assume is that all the memory in a failed protection domain is assumed to be corrupted—we assume a strong model of memory-based fault propagation. This model reflects the worst-case software behavior and prevents further corruption. An example of this is the notion of *undefined behavior* in language implementations such as C. Undefined behavior denotes semantic behavior in a language that is compiler-specific and often arbitrary. For example, writing past the end of an array in C is undefined behavior that can overwrite contiguous allocated memory, even if it is from a different and important data structure. Upon such a detected failure, often the only course of action is to restart code in the failed protection domain to bring it back into a consistent state. Fine-grained protection domains ensure that the scope of such failures is small. We do not focus on the fault propagation effects due to IPC from or to a failed component; instead we focus on the prevention of direct propagation via memory accesses. This has the effect of increasing the chances of detecting a failure soon after it occurs.

Based on the argument that finer grained memory isolation is beneficial to reliability, we focus on maximizing the memory-based fault isolation of the system by providing fine-grained protection domains. This causes inherent performance overheads as any communication between separate protection domains requires switching CPU page-tables. Given that this type of protection domain segregates memory regions, this paper focuses only on increasing memory isolation. We will use *isolation* as shorthand for *memory isolation* for brevity.

Detection of faults can be aided by hardware mechanisms (e.g., a page-fault hander or memory accesses outside of the allowed ranges), or done by software that detects when protocols are not adhered to (e.g., when code in a protection domain causes a deadlock or double-frees a memory allocation). Given memory isolation and the ability to detect errors, much research has been done on *recovery models* in the face of failures. Mechanisms for recovery from a failure within a protection domain include fine-grained transactions [35], recovering client data separately [11], saving and replaying client requests [37], and microrebooting individual components [6]. These techniques are essential and can be used in COMPOSITE, but are orthogonal to the fundamental goal of fine-grained fault isolation upon which these recovery techniques often rely. These techniques all require separate protection domains that enable memory isolation and attempt recovery on the granularity of a protection domain. Thus, we focus on minimizing the scope of the effected state from errant behavior by targeting fine-grained protection domains.

In this paper, we describe MPD in the context of our COMPOSITE component-based operating system. We consider a webserver application to demonstrate the extent to which MPD is able to maintain high performance and reliability. A server empowered by this mechanism ordinarily executes with the highest possible fault-isolation properties (i.e., many protection domains containing small pieces of functionality). However, under high-load conditions (e.g., when the responsiveness to HTTP requests falls below a certain threshold, or when availability is at risk), a system with MPD strategically removes protection boundaries, effectively widening the possible scope of memory accesses to maintain the highest fault isolation when ensuring performance constraints are met. In this situation, only those isolation boundaries with the most overhead and limited impact on reliability are removed. When load again decreases, MPD reconstructs protection boundaries and constrains the scope of memory accesses. In recognition of security constraints between specific components, certain protection boundaries can be treated as immutable.

In summary, the arguments in this paper center around the following concepts:

- Fine-grained fault isolation is beneficial to system reliability as it limits memory-based fault propagation and increases the ability to detect faults. Complementarity, recovery mechanisms that rely on this memory isolation, reestablish a consistent system state. However, finer grained fault isolation often increases performance overhead. Given this tradeoff, each system must determine how much

performance degradation is acceptable to achieve a certain fault isolation.

- The communication cost between specific components is often difficult to predict. However, this information is essential for system designers to determine where to place protection domains to best make this tradeoff. We observe that the communication patterns between components change over time. The subsystems of an OS that is heavily used are dependent on the currently executing application set, which is dynamic. Even within a single application, different inputs (a webserver handling normal or pipelined HTTP requests) cause significant changes in the amount of communication between certain components. Communication patterns change with differing systems loads as well: More frequent critical section access will cause more synchronization, thus producing more communication between synchronization and scheduling portions of the system. The inherently *dynamic* nature of the communication patterns in the system makes the design-time decisions about protection domain placement difficult. We show in Section 5.2 that by dynamically adapting protection domains, a performance level can be maintained while at any point in time providing more memory isolation barriers than a static configuration.

MPD moves the decision about how protection domains should be configured from design-time to runtime when the dynamic behaviors of the system are evident. In doing so, it aims to maximize the number of protection barriers in the system, while still meeting the performance constraints of the system. While dynamic reconfiguration is the focus of this paper, MPDs could also be used 1) to allow system designers to execute their system under expected workloads and have the system determine an appropriate protection domain configuration to be used statically, or to 2) aid in debugging via enhanced fault detection.

The rest of this paper is structured as follows: Section 2 describes the basis for MPD by discussing the design and implementation of components and their invocations in COMPOSITE. This is followed by a detailed description of the mechanisms used to control MPD in Section 3. Section 4 describes a component-based webserver to investigate the feasibility of MPD, and experiments are conducted on this system in Section 5. Section 6 presents the related research and Section 7 concludes.

2 COMPOSITE: COMPONENTS AND INVOCATIONS

To demonstrate MPD, we have developed a component-based system called COMPOSITE. A component in this context is a redeployable implementation of some functionality in accordance with an interface it exports to other components, and a collection of dependencies on other interfaces that are required for its execution [39]. Each system resource management policy and abstraction is defined as components, and a functional system is composed from a collection of such components. By default each component is isolated in its own protection domain. We say that a protection boundary is *collapsed* or *removed* between two components if they are placed into the protection domain, thus removing memory isolation between them. We say that a protection boundary is *raised* between components if they share a single protection domain, then are moved to separate protection domains and can no longer access each other's memory.

Fine-grained component isolation can lead to high communication costs. Motivated by this, μ-kernel researchers have achieved significant advances in Inter-Process Communication (IPC) efficiency [4], [28], [19]. Unfortunately, the hardware costs of performing IPC between protection domains on modern hardware can be significant, placing a significant lower bound on IPC overhead. COMPOSITE focuses not only on the traditional constraint that invocations between components in separate protection domains must incur little overhead, but also on efficient intraprotection domain invocations and the ability to dynamically switch between the two modes. We focus on the capability to *dynamically* alter the protection domain configuration of the system as different system inputs or concurrently executing applications will result in different communication overheads between specific components which change over time. For example, as we will discuss later, a webserver handling HTTP 1.0 or HTTP 1.1 pipelined requests imposes different communication patterns between components, as do requests for static content versus dynamic (CGI-generated) content. An operating system that supports many different applications will demonstrate different "hot-paths" between components depending on how different applications use the system, thus requiring the dynamic adaptation of the protection domain configuration to these specific overheads. The dynamic reconfiguration of protection domains is transparent to components and does not require their interaction. This section discusses the mechanisms COMPOSITE employs to provide efficient component invocations.

COMPOSITE employs a migrating thread model [17] in which the same schedulable entity executes across component boundaries. Invocation between components is depicted in Figs. 1a and 1b. In COMPOSITE, communication is controlled and limited by capabilities [26]. The presence of an unforgeable kernel-level capability structure signifies authorization for a component, c_0, to invoke a specific function in c_1. A corresponding "user-level capability" structure shared between user- and kernel-level contains a function pointer denoting the invocation method for the function in c_1. The kernel maintains strict access control as protection-domain crossings must be carried out via a capability in the kernel.

If intraprotection 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 interprotection domain invocation is required, a stub is invoked that marshals the invocation's arguments. The current COMPOSITE implementation supports passing up to four words 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 capability being invoked, the destination component (c_1),

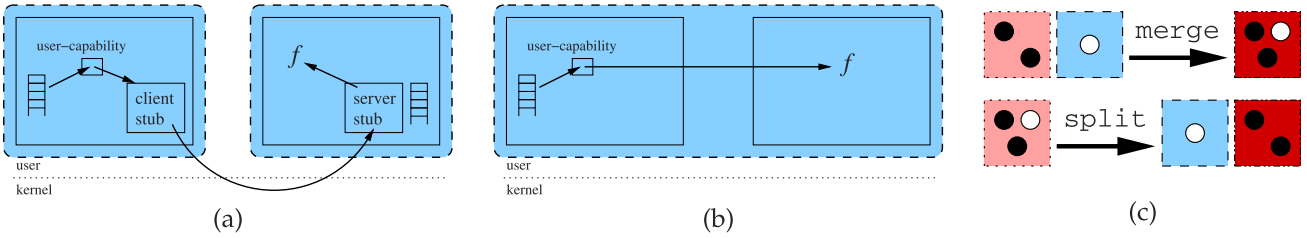

Fig. 1. (a) and (b) Invocation methods between components (solid boxes). (a) Invocations between protection domains (shaded, dashed boxes). (b) Intraprotection domain invocations. (c) MPD primitive operations.

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. A call stack is maintained for each thread to keep track of invocations between components. These invocation styles are depicted in Fig. 1. The essence of dynamically switching between inter and intraprotection domain invocations requires the kernel to 1) change the function pointer in the user-level capability structure from a direct pointer to c_1's function to the appropriate stub, and 2) appropriately manipulate the protection domains.

These mechanisms provide the foundation for trading off fault isolation for performance by concurrently enabling efficient invocations (both intra and interprotection domain), and the ability to quickly change between these two invocation methods.

An implication of the COMPOSITE component invocation design is that all components that can possibly be dynamically collapsed into the same protection domain must occupy nonoverlapping regions of virtual address space, as in single address space OSes [9]. As we will see in Section 3.3, the number of components that can share a virtual address space is limited not only by the size of that namespace, but also by architecture features (i.e., relating to page-tables). In COMPOSITE, this is not prohibitive because: 1) 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, 2) if components grow to the extent that they exhaust the virtual address space or otherwise cannot share the namespace, 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 3) where applicable, 64-bit architectures provide an address range that is large enough that sharing it is not prohibitive.

Though placing components into separate virtual address spaces, thus preventing protection boundaries from being removed between them, does in some sense force a static partitioning of the system, we believe that, practically, many of the benefits of MPD can be retained even when components must be spread across different virtual address spaces. Specifically, if components must be placed in different virtual address spaces due to (2), this can be done with some knowledge about where communication overheads between components were least significant in the past. Components at these communication boundaries are placed into separate virtual address spaces, thus lessening the likelihood of wanting to remove protection boundaries between these components in the future. We believe that separating components into different virtual address spaces is a rare operation as virtual address spaces can practically support many components. Thus, we leave these considerations as future work.

An operational COMPOSITE system includes a component that controls the mapping from components to protection domains, given communication overheads and performance targets. To aid in determining the overhead of communication between specific components, MPD uses counters to track how many invocations have been made between specific components. These counters avoid the overhead of using mechanisms such as hardware performance counters (e.g., `rdtsc` on x86) to measure communication costs.

3 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 to control the system's protection domain configuration that is used by an MPD policy component that decides system structure.

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 2, 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 frequently could prove quite expensive. One contribution of COMPOSITE is a design and implementation of MPD using portable hierarchical page-tables that is
   a. *transparent* to components executing in the system, and
   b. *efficient* in both space and time.

Section 3 discusses the primitive abstractions exposed to an MPD policy component used to control the protection domain configuration, and in doing so, reconcile MPD with component invocations and architectural constraints.

3.1 Semantics and Implementation of MPD Primitives

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 coexist in the new protection domain. This allows the MPD policy component to remove protection domain boundaries, and 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. This operation is depicted in Fig. 1c.

To increase the fault-isolation properties of the system, the COMPOSITE kernel provides the `split`(c_0) system-call. `split` removes the specified component from its protection domain and creates a new protection domain containing only 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 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 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 Fig. 1a). This operation is depicted in Fig. 1c.

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. To separate a protection domain containing multiple components into separate protection domains, each with more than one component, one component is split off, thus creating a new protection domain, and then the rest of the components are successively *moved* 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 expensive, both computationally and spatially. We investigate optimizations in Section 3.3.

3.2 Interaction between Component Invocations and MPD Primitives

Thread invocations between components imply lifetime constraints on protection domains. During an invocation to c_1, the memory of the invoking component (c_0) will be accessed (i.e., function arguments or the stack) if the components share the same protection domain *while executing in* c_1. Thus, if a protection barrier were erected while still executing in c_1 and accessing memory in c_0, the memory access to c_0 would cause faults. Unfortunately, these faults would be indistinguishable from erroneous behavior. We have a challenge in that raising protection boundaries and making intraprotection domain component invocations must in some way coexist, but the solution is not obvious. We consider two main options to enable the coexistence of intraprotection 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 removes the need for memory accesses in c_0 while executing in c_1, thus removing the conflict with MPD. This has the benefit of requiring no additional kernel support as the above problem is avoided, but significantly degrades invocation performance (relative to a direct function call with arguments passed as pointers).
2. 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. In this scheme, a protection domain configuration is maintained until threads making intraprotection domain invocations that would be effected by the MPD change return to c_0. This approach adds no overhead to component invocation, but requires 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 is important that component invocation overheads are removed by the system when necessary. If communication overheads cannot be completely removed, designers will be compelled to implement multiple abstractions or policies in the same component, thus weakening both system extensibility and reliability. Thus, the overhead of intraprotection domain component invocations should be on the order of a C function call. Appropriately, then, COMPOSITE uses the second approach to maintain efficient intraprotection domain invocations, and investigates if an implementation of intelligent MPD primitives is possible and efficient on commodity hardware using page-tables.

Toward this, 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*. The problem arises when a thread τ enters a protection domain A containing two components c_0 and c_1. If c_0 and c_1 were `split` into separate protection domains, B and C, τ's invocation from c_0 to c_1 could cause memory faults while accessing c_0 as discussed above. Taking this into account, COMPOSITE explicitly tracks the lifetime of thread's access to protection domains using reference counting. When a thread τ enters a protection domain A, a reference to A is taken, and when τ returns, the reference is released. Only if there are no other references to A is it freed. Thus, even though B and C are created, because τ maintains a reference to A it will continue to execute in A until it returns. The *current configuration* of protection domains all maintain a reference to prevent deallocation. In this way, the lifetime of protection domains accommodate 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` (A in this example)—remain active for τ.

Discussion. Efficient intraprotection 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 intraprotection domain invocations maintain high performance: Section 5.1 reveals their overhead to be on the order of a C++ virtual function call.

3.3 MPD Optimizations

The MPD primitives are used to remove performance hot-paths in the system. If this action is required to meet critical task deadlines in an embedded system or to increase performance before a server goes into overload, it must be performed in a bounded and short amount of time. Additionally, in systems that switch workloads and communication hot-paths 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 3.1, the implementation of `merge` and `split` are not practical. Each operation allocates new page-tables and copies sections 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 4 MB of memory. In a normal kernel, the resource management and performance implications of this allocation and copying are detrimental. For simplicity and efficiency reasons, the COMPOSITE kernel is nonpreemptible. Allocating and copying complete page-tables in COMPOSITE, then, is not practical. This problem is exacerbated by 64-bit architectures with deeper page-table hierarchies. Clearly, there is motivation for the OS to consider a more careful interaction between MPD and hardware page-table representations. During the initial development of MPD, we considered using a static scheme that cached protection domain configurations and allowed the system to switch between them. However, the current implementation provides generality, freedom over the placement of protection domains (instead of being constrained to a predefined set of structures), and performance. Thus, we found that there was no need to resort to more restrictive mechanisms.

Sharing page-table structures. 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. Fig. 2 shows three protection domain configurations: an initial configuration containing both c_0 and c_1, A, and the two resulting from a `split`, B and C. The top two levels correspond to the page-table structure, and the bottom to the component data pages. Each different protection domain configuration requires a page of memory

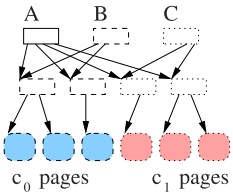

Fig. 2. COMPOSITE page-table optimization.

for the top level of the page-tables, and a 32 byte kernel structure describing the protection domain that contains a pointer to its page-table. Therefore, to construct a new protection domain configuration (via `merge` or `split`) requires allocating and copying only a page.

Optimizing `merge`. In addition to sharing second level page-tables, COMPOSITE further optimizes each primitive. We design `merge` to require no memory allocation (the "out-of-memory" case is not predictable in general, and memory allocation can be expensive). To optimize `merge`, 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. With this optimization, `merge` requires no memory allocation (indeed, it frees a page), and requires copying only B's component's entries to the top level of A's page-table.

Optimizing `split`. 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 its 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 protection domains, each containing multiple components, successive `splits` and `merges` 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 Section 5.1.

3.4 Mutable Protection Domain Policy

The 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 a policy that decides where protection domain boundaries should exist given communication patterns in the system. Our policy focuses on maximizing the number of protection domains in the system while meeting some performance goal. This policy attempts to minimize the fault propagation due to stray memory writes between components.

Though we focus on minimizing the number of components in shared protection domains, other policies might be of interest. These include minimizing the number of source lines of code in each protection domain, or minimizing the number of "risky" components—ones that have failed in the past, or have not been rigorously tested—that share protection

domains with other components. These would require appropriate changes to the policy of this section. Whichever policy is chosen, the overall goal is to adapt the protection domain configuration of the system to maximize some target for reliability, while achieving acceptable performance (e.g., to ensure availability). We leave a more thorough investigation of different policies for trading fault isolation for performance to future work.

In [33], a policy is introduced for finding a protection domain configuration given invocations between components and its effects are simulated on the system. We adapt that policy to use the proposed primitives. A main conclusion of [33] 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 used in COMPOSITE targets a threshold for the maximum number of interprotection domain invocations over a window of time. Thus, the 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 uses a min-cut algorithm [36] 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 groups of boundaries between components are tracked in separate priority queues. When the policy wishes to remove invocation overheads, the most expensive interprotection domain edge is chosen, and when the policy wishes to construct isolation boundaries, the min-cut at the head of the queue is chosen.

4 Application Study: Webserver

To investigate the behavior and performance of MPD in a realistic setting, we present a component-based implementation of a webserver that serves both static and dynamic content (i.e., using 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 pipelined through one connection. The components that functionally compose to provide these services are represented in Fig. 3. Each node is a component, and edges between nodes represent communication capabilities. Each name has a corresponding numerical id that is 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 webserver. Components related to CGI processing have two numerical ids in Fig. 3. The system includes two separate CGI programs represented as a shared set of components here. One of the CGI programs include components with ids 18, 20, and 21, and the other has 22, 23, and 25. We refer to the different CGI programs as `CGI A` and `CGI B`. Here too, the component graph could be much more complex as there could be an arbitrarily large number of different CGI programs.

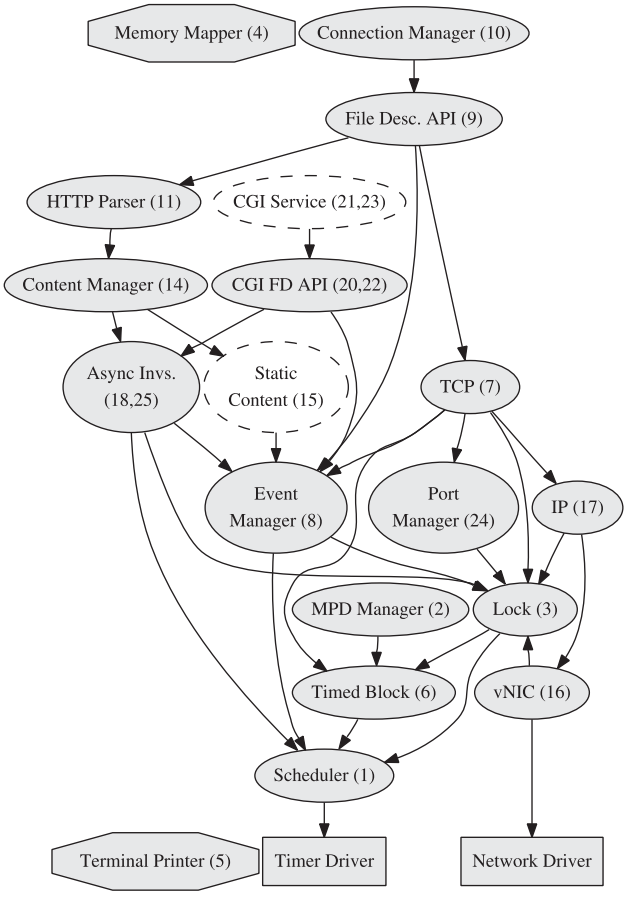

Fig. 3. COMPOSITE component-based webserver.

4.1 Webserver Components

We describe how the server is decomposed into components.

4.1.1 Thread Management

Scheduler: COMPOSITE has no in-kernel scheduler, instead relying on scheduling policy being defined in a component

at user level [34]. 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., TCP timers).

Lock: Provide a mutex abstraction for mutual exclusion. A library loaded into client components implements the fast-path of no contention in a manner similar to futexes [18]. Only upon contention is the lock component invoked.

Event Manager: Provide edge-triggered notification of system events in a manner similar to [3]. Threads are blocked to wait on inactive events. Producer components trigger events.

4.1.2 Networking Support

vNIC: COMPOSITE provides a 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 lwIP [29] port that provides 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. TCP requests an unused port for new connections, and releases them on connection termination.

4.1.3 Webserver 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 bidirectional and request/response based. This allows CGI components to be scheduled separately from the main application 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 (e.g., streams of HTTP data).

4.1.4 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 [16] webserver extensions.

4.1.5 Other Components

The *Memory Mapper* has the capability to map physical pages into other component's protection domains, thus controlling memory allocation. The *Terminal Printer* prints strings to the terminal. The debugging components are not shown: *Stack Trace* and *Statistics Gatherer*.

4.2 Webserver Data-Flow and Threads

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 an 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*. This 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 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 [32]. The COMPOSITE kernel is loaded as a module in Linux, and at the hardware interrupt/exception level, it takes over the system. Specifically, it overrides in the hardware tables and registers—the entry points for system-calls and faults. Thus, the COMPOSITE kernel essentially executes directly on the hardware by completely redefining system-call and interrupt handlers. However, we allow Linux to maintain interrupt handlers for device drivers (e.g., for the networking device). When packets arrive, they are processed by the device drivers in Linux, and then handed off to the COMPOSITE kernel to be processed by components.

5.1 Microbenchmarks

`Merge` and `split` overheads. First, we measure the overheads of the `merge` and `split` operations. To obtain these measurements, we execute the `merge` operation on

TABLE 1
MPD Primitive Operations

Operation	μ-seconds
`merge`	0.6
`split` w/ protection domain reuse	1.4
`split`	2.1
Clone address space in Linux	30.7

TABLE 2
Component Communication Operations

Operation	μ-seconds
Inter-PD component invocation	0.7
Hardware overhead of switching between two PDs	0.5
Intra-PD component invocation	0.008
Linux pipe RPC	6.4

10 components, first c_0 with c_1, then the resulting protection domain with c_2, and so on. We measure the execution time of this. Then, we `split` the components one at a time (first c_9 from the rest, then c_8, and so on), again measuring execution time. This is repeated 100 times, and the average execution time for each operation 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.

As a point of reference, we also include the cost of cloning an address space in Linux. In Linux, this operation is mature and efficient and 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 with 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. This measurement is included not as a competitor to MPD, as cloning an address-space is not semantically similar to MPD operations, but as a frame of reference for the performance of virtual address-space manipulations are in an optimized general-purpose system.

Table 1 presents the overheads of the primitive operations for controlling MPD in COMPOSITE. 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. 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.

Invocation performance. The efficiency of communication and coordination between components becomes increasingly important as the granularity of components shrinks. 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 reliability of the system.

Table 2 includes microbenchmarks of invocation costs between components. Each measurement is the result of the average of 100,000 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 0.7 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 inlined 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 0.5 μ-seconds. This is as close as we can get to removing all software overheads for an invocation and represents a lower bound on this processor. With 31 percent 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 [27], 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 0.08 μ-seconds, or 20 cycles. This overhead is on the order of a C++ virtual function call which carries an overhead of 18 cycles (using `g++` version 4.1.2).

We include the costs of an 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.

5.2 Apache Webserver Comparison

In measuring webserver performance, we use two standard tools: the Apache benchmark program (`ab`) [1] version 2.3, and `httperf` [20] version 0.9.0. We compare against Apache [1] version 2.2.11 with logging disabled and FastCGI [16] provided by `mod_fastcgi` version 2.4.6.

Here, we study the throughput of the webserver in COMPOSITE with an emphasis on the effect of protection domains. We also test the throughput of Apache in three different configurations that made different performance/fault isolation tradeoffs. Though Apache and COMPOSITE are of different levels of maturity, we provide both results for perspective. Fig. 4 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

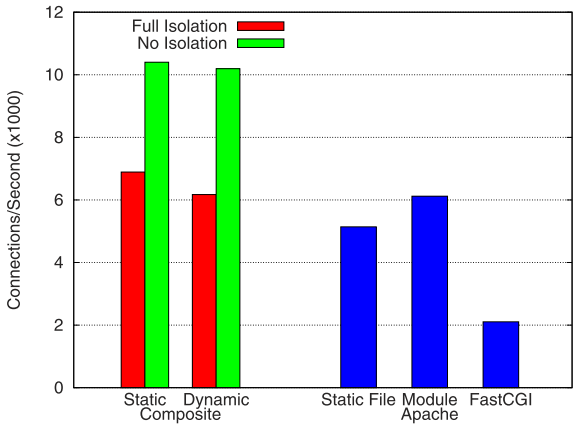

Fig. 4. Webserver throughput comparison.

COMPOSITE configurations. Requests for static content yield 6,891.50 connections/second with each component in a separate protection domain, and 10,402.72 connections/second when every component shares the same protection domain. Serving dynamic CGI content yields 6,170.74 and 10,194.28 connections/second for full and no isolation, respectively. For Apache, we find that 20 concurrent connections maximizes throughput for serving a (cached) static file at 5,139.61 connections/seconds, 32 concurrent connections maximizes throughput for module-generated content at 6,121.27, and 16 concurrent connections maximizes throughput at 2,106.39 connections/second for fastCGI dynamic content. All content sources simply return an 11 character string.

The three Apache configurations demonstrate design points in the tradeoff between dependability and performance. Apache modules are compiled libraries loaded directly into the server's protection domain. This minimizes 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 webserver 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 communication costs are high.

The comparison between COMPOSITE and Apache is not straightforward. On the one hand, Apache is a much more full-featured webserver than our COMPOSITE version (e.g., it supports more HTTP protocol options, security policies, and supports multiple means of interacting with CGI scripts), which could negatively effect Apache's throughput. On the other hand, Apache is a mature product that has been highly optimized. COMPOSITE achieves higher throughput, partially because it is less feature-rich. We compare against Apache to investigate the potential of practical COMPOSITE performance. Thus, we propose the most interesting conclusion of these results is the 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 percent by removing protection boundaries, thus sacrificing dependability.

5.3 The Tradeoff between Fault Isolation and Performance

Next, we investigate the tradeoff between fault isolation and performance, and specifically the extent to which isolation must be compromised to achieve significant performance gains. Fig. 5a presents two separate scenarios in which the server handles 1) static content requests and 2) dynamic CGI requests, (generated with `ab`). 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 six protection domains have been merged, throughput exceeds 90 percent of its maximum. For dynamic content, when eight protection domains are merged, throughput exceeds 90 percent. The critical path of components for handling dynamic requests is longer than that for static content 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, Figs. 5b and 5c 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. Fig. 5b represents the system while processing static requests. Fig. 5c represents the system while processing dynamic requests using HTTP 1.1 persistent connections (generated with `httperf`). In this case, 2,000 connections/second each make 20 pipelined `GET` requests.

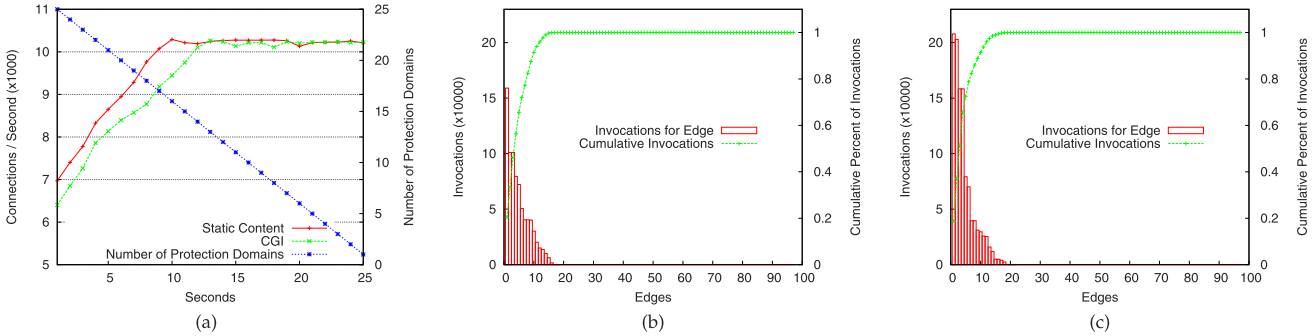

Fig. 5. (a) The effect on throughput of removing protection domains. (b) and (c) The number of invocations between components, and the CDF of the invocations for (b) HTTP 1.0 static content requests, (c) persistent HTTP 1.1 CGI requests.

TABLE 3
Hot-Path Edges from Figs. 5b and 5c

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

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 percent, respectively, of the component invocations. These results show promise for MPD as they imply that a small amount of protection boundaries need be removed to achieve large performance gains.

Table 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 hot-paths for the same system under different workloads differ greatly. If the system wishes to maximize throughput while merging the minimum number of protection domains, different workloads require significantly different protection domain configurations. This confirms the essence of the argument for the *dynamic* reconfiguration of protection domains to adapt to the overheads of specific, and changing workloads.

5.4 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 hot-paths in the system change over time. To investigate the effectiveness of MPD, Figs. 6 and 7 compare two MPD policies, 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 (i.e., merges protection domains), and the other both merges and splits (labeled in the figure as "Full MPD Policy"). The policy that

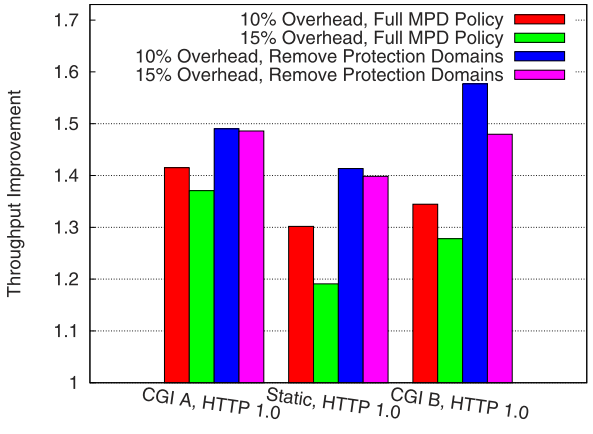

Fig. 6. The throughput improvement over full isolation.

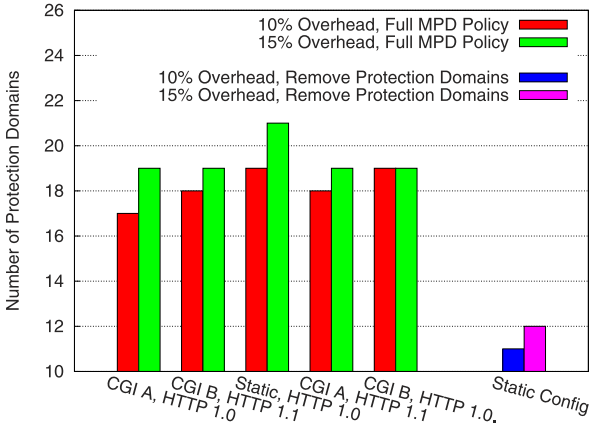

Fig. 7. The number of active protection domains.

only merges protection domains represents an oracle of sorts that we use as a proxy for an optimal system that uses a *static* protection domain configuration. If the system designer were to statically make the mapping of components to protection domains such that target performance constraints are always met, they would want to choose the mapping at the *end* of the experiments. This is because protection domain boundaries are only removed when performance goals are not made; thus we can be sure that the final configuration would meet all performance constraints, while having a maximal (according to the MPD policy) number of protection domains. This merge-only policy is an oracle as it uses runtime knowledge to produce its final "static" configuration. *We compare the final configuration of the merge-only policy with the full MPD policy that both merges and splits to investigate how effective* dynamic *reconfiguration of protection domains is.*

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 when only removing protection domain boundaries. 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 interprotection domain invocations is set to a calculated 10 percent of processing time, and the other with it set to 15 percent. 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. Ten percent corresponds to 142,857 invocations, and 15 percent corresponds to

214,285 invocations. These invocation counts approximate the allocated isolation overhead and do not capture cache effects that might change the final overhead. This is a very simple policy for managing the tradeoff between overhead and fault isolation. Certainly, more interesting policies taking into account task deadlines or other application metrics could be devised. Additionally, policies that give isolation between specific components different importance values or weights could allow the protection domain configuration to take into account isolation priorities between well-tested (thus likely less faulty) or prototype (thus more erroneous) components. 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.

Fig. 6 plots the throughput relative to a full isolation system configuration for different workloads. We do not 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 2,000 connections/second with 20 requests per connection.

All approaches maintain significant increases in performance. It is not surprising that the policies that only remove isolation increase performance over time. The full MPD policies improve performance, on average, by 35 and 28 percent for 10 and 15 percent fault-isolation overhead, respectively.

Fig. 7 plots the number of protection domains (25 being the maximum possible, 1 the minimum) in the system. 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 percent, respectively. This translates to between 68 (i.e., 17 out of 25) and 84 percent (i.e., 21 out of 25) for isolation overheads of 10 and 15 percent, 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 hot-paths of differing workloads by maintain higher levels of isolation. They do this while still achieving significant performance gains that correspond to a system-decided tradeoff between fault isolation and performance.

Qualitatively, Table 4 represents the protection domain configuration for three of the workloads and different MPD policies. Each group of comma-separated components surrounded by parentheses is coresident 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 policy that represents a possible static system configuration (remove PD only in the table) with a tolerance of 10 percent overhead has merged 14 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.

Importantly, with MPD, the system returns to a protection domain configuration with full fault isolation when the system load decreases, thus promoting reliability. Any static configuration that compromises fault-isolation boundaries pessimistically in anticipation of future overhead, cannot benefit system reliability when system load is low.

TABLE 4
Protection Domain Configurations Resulting from Different Workloads and Different MPD Policies

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)

MPD policy overhead. In addition to the MPD primitive operation's microbenchmarks in Section 5.1, here we report the processing costs of executing the MPD policy in the more realistic multiworkload environment. The MPD policy is run four times a second to manipulate the protection domain configuration, and the total processing time for this is recorded each second. Throughout the course of the experiment, 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.

6 Related Work

Many researchers have proposed numerous fault-isolation approaches, in both software and hardware. Type-safe languages [21] often use dynamic checks to avoid the arbitrary corruption of memory. When a fault is detected, it is often not straightforward to determine the full extent of the possible corruption [2]. Some systems explicitly track the modified state by a path of execution, thus allowing a transaction-style roll-back in the case of error [35]. Such techniques impose performance overheads that limit the granularity of fault isolation. In other systems, software is segregated into explicit servers [14] that share no memory. Communication between these servers is conducted via IPC, which still imposes a significant performance overhead (of the same magnitude as invocations in COMPOSITE), which motivates MPD. An implementation of MPD in such systems might be possible and is an area of future investigation. A benefit of using hardware techniques to provide protection domains is the ability to execute machine code components, thus supporting the widest breadth of software including legacy code.

Hardware techniques have long been used to provide fault isolation. Page-tables are a portable structure for

defining protection domains, but systems have provided extensions for more accurate [40] protection mechanisms. As the granularity of the protection domains decreases, the performance of interprotection domain communication becomes increasingly important. Many systems have contributed to increased IPC performance [13], [4], [27] by focusing on a simplification of the system's core abstractions. These approaches still incur a processing overhead for high rates of communication that can adversely affect the system's ability to meet performance goals. Such systems, then, still motivate the ability to dynamically tradeoff fault isolation for performance.

Systems that increase the designer's ability to find the hot-paths in the system [7] help the designer to alleviate system bottlenecks. Such mechanisms could potentially be used to inform a designer where to remove isolation boundaries to achieve performance goals. Unfortunately, such tools are useful only for the *profiled workload*, and not all possible workloads and applications that might execute on a system. MPD provides detailed information to identify system hot-paths to the MPD policy. The MPD policy then uses the primitives for controlling MPDs to automatically tradeoff fault isolation for performance exactly where that sacrifice will most improve system performance. Enabling system code to make this tradeoff in accordance with its own policy allows it to adapt to dynamic and unpredictable system execution.

7 Conclusion

This paper introduces Mutable Protection Domains that enable the system to control the tradeoff between fault isolation and performance. Conceptually, MPD presents fault isolation as a system property that is dynamically manipulated to alter system throughput or latency on demand.

We present an implementation of MPD in the COMPOSITE component-based operating system that is efficient both in terms of space and time. We discuss the difficulties of implementing MPD using common architectural structures such as page-tables, and of simultaneously providing an efficient component invocation path. Additionally, we evaluate MPD with a novel component-based webserver. Results show that MPD increases throughput for our workloads by up to 40 percent over a system with full isolation while still maintaining as much as 84 percent highest fault isolation.

In the future, we wish to investigate MPD policies that take into account the perceived trustworthiness of specific components in calculating system structure. Additionally, in this paper we focus on providing the smallest possible fault isolation domains with MPD under the assumption that finer grained fault containment enables easier system recovery. We wish to continue this investigation by applying known recovery techniques to transform the increased fault isolation into increased fault tolerance.

References

[1] Apache Server Project, http://httpd.apache.org/, 2012.

[2] G. Back and W.C. Hsieh, "Drawing the Red Line in Java," *Proc. Seventh Workshop Hot Topics in Operating Systems,* 1999.

[3] G. Banga, J.C. Mogul, and P. Druschel, "A Scalable and Explicit Event Delivery Mechanism for UNIX," *Proc. USENIX Ann. Technical Conf.,* 1999.

[4] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy, "Lightweight Remote Procedure Call," *ACM Trans. Computer System,* vol. 8, no. 1, pp. 37-55, 1990.

[5] C. Cadar, D. Dunbar, and D.R. Engler, "KlEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *Proc. Eighth USENIX Conf. Operating Systems Design and Implementation,* 2008.

[6] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot—A Technique for Cheap Recovery," *Proc. Sixth Conf. Symp. Operating Systems Design and Implementation,* 2004.

[7] B.M. Cantrill, M.W. Shapiro, and A.H. Leventhal, "Dynamic Instrumentation of Production Systems," *Proc. USENIX Ann. Technical Conf.,* 2004.

[8] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, "Hive: Fault Containment for Shared-Memory Multiprocessors," *SIGOPS Operating Systems Rev.,* vol. 29, no. 5, pp. 12-25, 1995.

[9] J.S. Chase, M. Baker-Harvey, H.M. Levy, and E.D. Lazowska, "Opal: A Single Address Space System for 64-Bit Architectures," *ACM SIGOPS Operating Systems Rev.,* vol. 26, no. 2, pp. 80-85, 1992.

[10] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating Systems Errors," *Proc. 18th ACM Symp. Operating Systems Principles,* 2001.

[11] F.M. David, E.M. Chan, J.C. Carlyle, and R.H. Campbell, "CuriOS: Improving Reliability through Operating System Structure," *Proc. Eighth USENIX Conf. Operating Systems Design and Implementation,* 2008.

[12] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the Art of Virtualization," *Proc. 19th ACM Symp. Operating Systems Principles,* 2003.

[13] D.R. Engler, F. Kaashoek, and J. O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. 15th ACM Symp. Operating Systems Principles,* 1995.

[14] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G.C. Hunt, J.R. Larus, and S. Levi, "Language Support for Fast and Reliable Message-Based Communication in Singularity OS," *Proc. First ACM SIGOPS/EuroSys European Conf. Computer Systems,* 2006.

[15] J. Fassino, J. Stefani, J. Lawall, and G. Muller, "Think: A Software Framework for Component-Based Operating System Kernels," *Proc. Usenix Ann. Technical Conf.,* 2002.

[16] FastCGI, http://www.fastcgi.com, 2012.

[17] B. Ford and J. Lepreau, "Evolving Mach 3.0 to a Migrating Thread Model," *Proc. Winter USENIX Technical Conf.,* 1994.

[18] H. Franke, R. Russell, and M. Kirkwood, "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux," *Proc. Ottawa Linux Symp.,* 2002.

[19] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz, "The Pebble Component-Based Operating System," *Proc. Usenix Ann. Technical Conf.,* 2002.

[20] httperf, www.hpl.hp.com/research/linux/httperf/, 2012.

[21] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A Safe Dialect of C," *Proc. USENIX Ann. Technical Conf.,* 2002.

[22] P. Joubert, R.B. King, R. Neves, M. Russinovich, and J.M. Tracey, "High-Performance Memory-Based Web Servers: Kernel and User-Space Performance," *Proc. USENIX Ann. Technical Conf.,* 2001.

[23] A. Lenharth, V.S. Adve, and S.T. King, "Recovery Domains: An Organizing Principle for Recoverable Operating Systems," *Proc. 14th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* 2009.

[24] J. Lepreau, M. Hibler, B. Ford, J. Law, and D.B. Orr, "In-Kernel Servers on Mach 3.0: Implementation and Performance," *Proc. Third Conf. USENIX MACH III Symp.,* 1993.

[25] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines," *Proc. Sixth Conf. Symp. Operating Systems Design and Implementation,* 2004.

[26] H. Levy, *Capability-Based Computer Systems.* Digital Press, 1984.

[27] J. Liedtke, "Improving IPC by Kernel Design," *Proc. 14th ACM Symp. Operating Systems Principles,* 1993.

[28] J. Liedtke, "On Micro-Kernel Construction," *Proc. 15th ACM Symp. Operating System Principles,* 1995.
[29] lwIP, http://www.sics.se/~adam/lwip/index.html, 2012.
[30] NIST Study—The Economic Impacts of Inadequate Infrastructure for Software Testing, http://www.nist.gov/director/prog-ofc/report02-3.pdf, 2012.
[31] J. Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware?" *Proc. Summer USENIX Conf.,* 1990.
[32] G. Parmer and R. West, "Hijack: Taking Control of COTS Systems for Real-Time User-Level Services," *Proc. IEEE 13th Real Time and Embedded Technology and Applications Symp.,* 2007.
[33] G. Parmer and R. West, "Towards a Component-Based System for Dependable and Predictable Computing," *Proc. IEEE 28th Real-Time Systems Symp.,* 2007.
[34] G. Parmer and R. West, "Predictable Interrupt Management and Scheduling in the Composite Component-Based System," *Proc. Real-Time Systems Symp.,* 2008.
[35] M.I. Seltzer, Y. Endo, C. Small, and K.A. Smith, "Dealing with Disaster: Surviving Misbehaved Kernel Extensions," *Proc. Second USENIX Symp. Operating Systems Design and Implementation,* 1996.
[36] M. Stoer and F. Wagner, "A Simple Min-Cut Algorithm," *J. ACM,* vol. 44, no. 4, pp. 585-591, 1997.
[37] M.M. Swift, M. Annamalai, B.N. Bershad, and H.M. Levy, "Recovering Device Drivers," *Proc. Sixth Conf. Symp. Operating Systems Design and Implementation,* 2004.
[38] M.M. Swift, B.N. Bershad, and H.M. Levy, "Improving the Reliability of Commodity Operating Systems," *Proc. 19th ACM Symp. Operating Systems Principles,* 2003.
[39] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., 2002.
[40] E. Witchel, J. Cates, and K. Asanović, "Mondrian Memory Protection," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* 2002.

Gabriel Parmer is working as an assistant professor of computer science at The George Washington University. His research interests include operating systems, component-based systems, embedded systems, and real-time systems. His work revolves around COMPOSITE, a component-based OS. He focuses on building a practical, predictable, customizable, and dependable system. He is a member of the IEEE.

Richard West is working as an associate professor of computer science at Boston University. He holds an advanced research engineer position with VMware, where he consults on resource management techniques for hypervisors applied to multicore architectures. His research interests encompass operating systems, real-time systems, and resource management. His work includes the development of a new operating system that is predictable and safe and leverages hardware performance counters to increase efficiency in the presence of microarchitectural resource contention. He is a member of the IEEE.