Mondrix: Memory Isolation for Linux using Mondriaan Memory Protection

Emmett Witchel
Department of Computer Sciences
University of Texas at Austin
Austin TX 78712
witchel@cs.utexas.edu

Junghwan Rhee
Purdue University
West Lafayette, Indiana 47907
rhee@purdue.edu

Krste Asanović*
MIT Computer Science and Artificial Intelligence Laboratory,
Cambridge, MA 02139
krste@csail.mit.edu

ABSTRACT

This paper presents the design and an evaluation of Mondrix, a version of the Linux kernel with Mondriaan Memory Protection (MMP). MMP is a combination of hardware and software that provides efficient fine-grained memory protection between multiple protection domains sharing a linear address space. Mondrix uses MMP to enforce isolation between kernel modules which helps detect bugs, limits their damage, and improves kernel robustness and maintainability. During development, MMP exposed two kernel bugs in common, heavily-tested code, and during fault injection experiments, it prevented three of five file system corruptions.

The Mondrix implementation demonstrates how MMP can bring memory isolation to modules that already exist in a large software application. It shows the benefit of isolation for robustness and error detection and prevention, while validating previous claims that the protection abstractions MMP offers are a good fit for software. This paper describes the design of the memory supervisor, the kernel module which implements permissions policy.

We present an evaluation of Mondrix using full-system simulation of large kernel-intensive workloads. Experiments with several benchmarks where MMP was used extensively indicate the additional space taken by the MMP data structures reduce the kernel's free memory by less than 10%, and the kernel's runtime increases less than 15% relative to an unmodified kernel.

Categories and Subject Descriptors

D.4.5 [**Operating systems**]: Reliability

General Terms

Reliability

Keywords

fine-grained memory protection

*This work was partly supported by NSF CAREER Award CCR-0093354.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'05, October 23–26, 2005, Brighton, United Kingdom.
Copyright 2005 ACM 1-59593-079-5/05/0010 ...$5.00.

1. INTRODUCTION

Reliability and security are quickly becoming users' biggest concerns due to the increasing reliance on computers in all areas of society. Operating systems written in unsafe languages are efficient, but they crash too often and are susceptible to malicious attacks. Crashes and security breaches incur large costs in lost productivity and increased system administration overhead. Many of these incidents could be reduced in severity or even avoided, if a fault in a single software module was caught before it propagated throughout the system. Faults often lead to illegal memory accesses, and wild writes can cause further modules to fail. Memory isolation, which forbids one software module from reading or writing another module's memory without permission, is therefore a crucial component of a robust system.

Mondriaan Memory Protection (MMP) [43] is a recently proposed fine-grained memory protection scheme that provides word-granularity memory isolation in hardware. Previous work investigated the use of MMP for user-level applications [43] and sketched how an operating system might employ MMP [42]. In this paper, we present the design and evaluation of Mondrix, a version of the Linux 2.4.19 kernel enhanced with MMP to provide memory isolation between kernel modules. Mondrix runs on top of versions of the SimICS [27] and Bochs [23] system simulators, which are modified to model MMP hardware. The main contributions of this paper are:

- The design and implementation of a fine-grained kernel memory protection system. A small module containing interfaces to MMP hardware and the permissions tables forms the most privileged layer and lives underneath the rest of the kernel. More complex permission abstractions and management policies are layered in separate higher-level modules.
- Implementation of a compartmentalized Linux kernel with eleven isolated modules, including ad-hoc modules already present in the Linux kernel such as unix domain sockets, a network device driver split into two modules, and a disk device driver split into three modules.
- Several modifications to the original MMP hardware design to improve cross-domain calls and stack permission handling.
- An evaluation of the performance and space overheads of the full Mondrix implementation for a range of kernel-intensive application workloads. The results show that Mondrix executes less than 15% more cycles (instructions and memory stalls) than an unmodified kernel, and its data structures reduce the amount of kernel free memory by less than 10%.

- Results from fault-injection experiments showing how Mondrix can catch errors before they cause data corruption. Five fault-injection experiments caused file corruption in Linux, and Mondrix prevented file system corruption in three of those five experiments.

One advantage of MMP hardware memory isolation is that it is compatible with existing legacy code written in unsafe languages. An alternative approach is to rewrite system code in a safe language, such as Java or C#, that use a combination of a strict type system and garbage collection to avoid a large class of memory access errors encountered at run time. For example, Microsoft is using safe languages [11] in their next generation Windows Vista operating system. Safe languages can incur large performance overheads and require unsafe extensions to interface to the lower levels of a machine (though such extensions can be used sparingly). High performance implementations of safe languages require optimizing compilers and run-time systems, which increases the amount of code that must be trusted.

Although MMP requires a hardware change, it is backwards compatible with existing instruction sets and compiled user applications. The recent introduction of the NX bit [2] to the x86 architecture by AMD and Intel indicates that manufacturers are willing to add compatible hardware features to improve software robustness. We focus on the application of MMP to a conventional operating system in this paper, but note that MMP is designed to provide protected sharing for many kinds of large extensible software systems, such as web browsers and web servers. Mondrix could be extended to exploit additional MMP features, for example to provide more efficient yet safe user-level interfaces to kernel data structures.

Preventing illegal memory accesses is not sufficient to guarantee system reliability and security. Other failure modes include API violations, excessive resource consumption, and synchronization or locking errors. New static analysis [28, 13, 3] techniques can help locate many of these other sources of software failure as well as some types of illegal memory access. But these analyses can sometimes find thousands of possible violations, overwhelming the ability of developers to fix them all. Also, most analyses are unsound, so they do not find all errors. Although these techniques are useful, they are complementary to dynamic checking of memory accesses.

Our fault injections experiments validate the usefulness of memory isolation in detecting failures and preventing failures from damaging system state. Mondrix is able to eliminate file system corruption in three of five cases, and it detects memory use violations in 90% of executions that lead to kernel panics.

The paper takes a bottom-up approach in describing the Mondrix design. We begin in Section 2 with a review of the primitives provided by the MMP hardware. We present new schemes to provide protected control transfer between domains and to manage stack permissions, influenced by our experience in developing Mondrix. Section 3 presents for the first time the design and implementation of the memory supervisor, a software layer that sits below the kernel and which uses the raw MMP hardware to provide a number of permissions abstractions for higher levels of software. Section 4 describes the modifications made to Linux, including how the kernel shares memory with protected modules for disk, networking, and other services. We also describe policies for managing protected memory regions and for handling interrupts safely. We used full-system simulation with a modified SimICS and Bochs x86 simulator to evaluate the performance of Mondrix (Section 5). Under a variety of kernel-intensive workloads, we observe less than a 10% reduction in kernel free memory, and a slowdown of less than 15%.

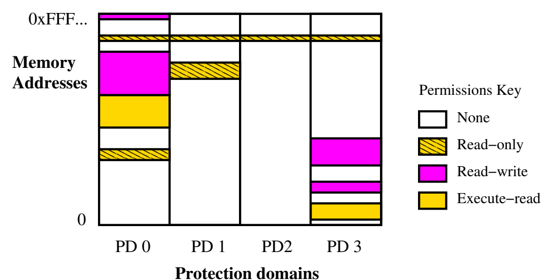

Figure 1: A visual depiction of multiple memory protection domains within a single shared address space.

We review related work in Section 6 before concluding.

2. MMP FEATURES

The three main features of MMP are memory protection, protected cross-domain calling, and stack protection. This section briefly reviews the main MMP features, and describes how MMP has been modified from the original design [43, 42] to support Mondrix.

2.1 Memory protection

MMP adopts Lampson's term [22], *protection domain*, to refer to a lightweight context that determines permissions for executing code. As shown in Figure 1, MMP overlays an address space with multiple disjoint protection domains, each with a unique set of permissions. Each column represents one protection domain, while each row represents a range of memory addresses. In Mondrix, the address space is the kernel virtual address space. There is no domain-specific portion of an address; a pointer refers to the same kernel memory location from any domain. Every thread is associated with exactly one protection domain at any point in its execution, and any number of threads may be in the same protection domain at the same time. The color in each box represents the permissions that a protection domain has to access the region of memory in the box. MMP allows any number of memory regions within a domain, and each region can begin and end at any word-aligned address.

The MMP implementation (Figure 2) stores compressed permissions information in permissions tables held in main memory, and caches the tables using an on-chip protection lookaside buffer (PLB) [21]. MMP hardware in the processor pipeline uses the PLB to check permissions on every load, store, and instruction fetch, and raises a protection exception if the executing thread does not have permissions for an attempted access. Implementing memory permission checks has limited impact on a typical out-of-order superscalar processor pipeline. The PLB is comparable in size and access latency to a conventional TLB, and the PLB hit rate is high [43]. Because permission checks are separate from address translation, data can be speculatively read and used before permissions checks are completed. Permissions checks need only be completed prior to final instruction commit. The permissions table is the only in-memory structure whose size is large and whose size scales with application memory use.

MMP preserves the user/kernel mode distinction, where kernel mode enables access to privileged control registers and privileged instructions. The CPU encodes whether a domain is user or kernel mode using the high bit of the PD-ID control register (a zero high bit implies a kernel domain). Protection domain 0 is used to manage the permissions tables for other domains, and can access

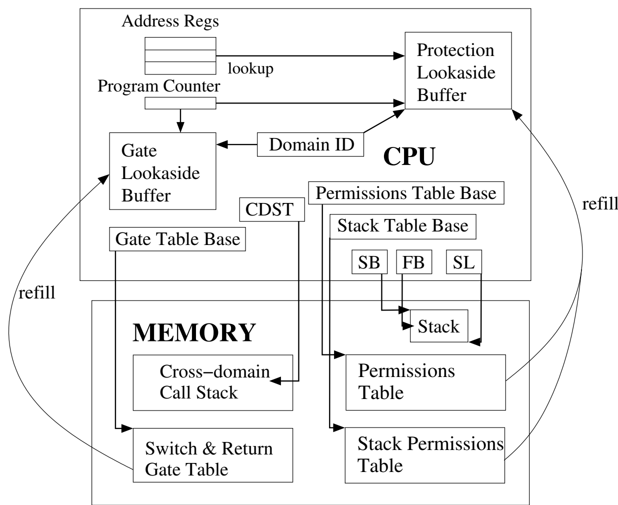

Figure 2: The major components of the Mondriaan memory protection system. On a memory reference, the processor checks permissions for the effective address in the protection lookaside buffer (PLB). In parallel, accesses are range checked with the registers that delimit permissions for a region of the stack: `sb`, `fb`, and `sl`. If permissions are not found in either check then hardware or software looks up the effective address in the memory-resident permissions table. The permissions come from the stack protection table if the miss address is a stack address, otherwise they come from the protection table. The reload mechanism caches the matching entry from the permissions table in the PLB. The gate lookaside buffer (GLB) caches information for cross-domain call entry sites held in the switch & return gate table. The `CDST` register points to the current top of the cross-domain call stack.

all of memory without the mediation of a permissions table. Only the bottom half of the memory supervisor (see Section 3) resides in PD 0.

2.2 Cross-domain calling

Cross-domain calling in MMP provides a two-way guarantee; first that a thread can only enter a callee's domain at specified points (*switch gates*), and second that a thread returning from a cross-domain call will return to the caller's domain only at the instruction following the call to the corresponding switch gate. The processor switches domains when a call instruction's target has a switch gate permission, or it executes a return instruction marked with a return gate permission. The programmer places switch gates on the first instruction of a routine (which is why they are called switch gates, not call gates), so call sites do not have to be identified when exporting a function, and a single indirect call instruction can call both exported and non-exported routines. The programmer places return gates on the return instruction of an exported routine.

Gates require more information than regular memory permissions, and so are stored in a separate gate table and cached with a separate gate lookaside buffer (GLB) (Figure 2). This is an improvement over the previous MMP design that did not dedicate a table to gates. The number of gates, even for a large system, is low (less than 1,000 in Mondrix), because modules tend to have many more internal functions than exported entry points. The gate table is stored in memory in an open hash table to allow rapid retrieval on a GLB miss. Each entry has the format shown in Figure 3. The first

Address (32b)		
Switch/Return (1b)	Unused (15b)	Destination PD–ID (16b)

Figure 3: The format of entries in the gate table. The gate table encodes cross-domain call points, with switch gates encoding the callee's protection domain.

word contains the byte address of the gate instruction. The second word of the entry specifies the gate type, and if it is a switch gate, the destination protection domain.

MMP gates are simpler than those present in the x86, IA-64 [10], or PA-RISC because they do not cause a stack switch. They are similar in spirit, though simpler in implementation to call and return capabilities (used in EROS [33]), and capabilities used in Multics [31]. By enforcing call/return semantics on cross-domain calls, MMP limits the possible implementations of exception mechanisms [30]. Exception management techniques are beyond the scope of this paper, but we note that call/return was sufficient to handle exception support for Mondrix.

The previous MMP design required hardware for the cross-domain call stack. The Mondrix version of MMP uses protected memory in the user area, that the hardware can write, but software (outside of the memory supervisor) can only read. Mondrix MMP retains the `CDST` register that points to the current top of the cross-domain call stack. The Mondrix memory supervisor saves and restores this register on a context switch.

For each cross domain call, the processor pushes the return address of the call instruction whose target is the switch gate, the current protection domain ID, and the current value of the `fb` register (an addition from previous work whose function is explained in the next section). These values are popped and verified on a cross-domain return. A domain can establish which domain called it by reading the cross-domain call stack, and it can trust the value because the cross-domain call stack is only writable by hardware.

One issue in Mondrix is cross-domain calls that do not change protection domain ID. The processor executes return gates in the callee's domain, which causes problems if a domain calls a function that it also exports. Consider, for example, `kmalloc`. The core kernel exports this routine to modules, so it must place a return gate on its last instruction. If the kernel were to call it via a regular function call, the instruction with the return gate would fault because a regular function call does not push the state needed for a cross-domain return onto the cross-domain call stack. Therefore a domain must either mark the entry points to exported functions with a switch gate, or it must duplicate exported functions. We chose to mark exported functions with a switch gate, avoiding the task of classifying function calls into domain-crossing and non-domain-crossing. Unfortunately, this decision has the consequence of more than doubling the number of cross-domain calls (Section 5.2.1).

Cross-domain calls require modifications to the processor hardware. Each instruction fetch checks the GLB for the presence of a gate. Each instruction cache line has an additional bit indicating if there are any gates associated with instructions on that line. If the bit is clear, then no further action needs to be taken on a GLB miss. If the bit is set and the GLB misses, the GLB must be refilled from the gate table. If the bit is set and there is a hit in the GLB, the cached gate is used. Instruction cache lines are initially brought in after a miss with the "gate present" bit set, but the bit is cleared down if a subsequent GLB miss and gate table walk determines there are no gates on the line.

2.3 Stack permissions

Stack storage must be protected differently from other memory, because stacks are associated with threads that can move between protection domains. The Mondrix MMP stack permissions design fixes problems in the previous MMP designs [42], where different threads resident in the same protection domain had access to each other's stacks. This issue is addressed by adding more per-thread hardware.

Mondrix maintains two parallel forms of thread-local stack permissions. Stack permission registers designate stack frames in the current domain as readable and writable (between the frame base register (`fb`) and the stack limit register `sl`), and earlier frames (between the stack base register `sb` and `fb`) as read-only. Stacks grow down, so the stack base register is at a higher address than the stack limit register. A separate stack write permissions table allows individual words of earlier stack frames (between `sb` and `fb`) to be thread-writable (see Figure 2). A stack location is writable if it lies between the read-only and read-write register addresses (`fb` and `sl`), or if its stack write permissions bit is set.

The stack registers and permission table support the common idioms of stack use. The registers allow read-write access to stack frames for the thread's execution in the current domain, and read-only access to previous frames. Stack accesses to the current frame and reads from previous frames are handled efficiently. The stack permissions table supports existing calling conventions with parameters that point to writable stack-allocated data structures. The stack write permissions table encodes whether a given stack address is writable by the thread, using one bit per word, and the contents of this table is cached in the PLB.

On a cross-domain call the hardware saves the current frame base on the cross-domain call stack, and the current stack pointer becomes the new frame base (unless the stack pointer points outside `sb` and `sl` in which case the processor faults). Cross-domain calls move all stack frames that were allocated in the previous domain into the read-only area between `sb` and `fb`.

The memory supervisor only allows a thread to grant write permissions on its current frame area, a thread may not grant itself write permissions on a previous domain's frames. The memory supervisor flushes stack permissions information from the PLB when a thread is descheduled, and also unloads and reloads a thread's stack permission register during context switches.

3. THE MEMORY SUPERVISOR

This section describes the features and implementation of the Mondrix memory supervisor, which was designed to easily slip "under" an existing kernel to form the most privileged software layer. The supervisor is split into two pieces, a top and a bottom. The bottom layer (which is not checked by MMP hardware) has the sole job of writing the permissions tables in memory. The top layer does everything else, including presenting a hardware-independent memory protection interface to the rest of the kernel, enforcing memory protection policies, tracking memory sharing, and implementing group protection domains.

This section describes the top layer of the supervisor. The purpose of Mondrix is to provide memory isolation, and the memory supervisor enforces policies about memory sharing, such as a domain can't give itself write permission on a piece of memory that was exported to it read-only. If the top half of the supervisor decides a permissions request is valid, it passes the request to the bottom half which updates the protection tables.

This section defines several terms (including access and ownership) for memory use, and then presents the intuitions behind the supervisor's policies for memory sharing, as well as a detailed summary of of the policy (Table 1).

3.1 Definitions for memory use

Access permissions

A domain's permission, gate, and stack tables jointly describe its *access permissions*, i.e., the operations it can perform on memory such as execute a return gate or write a location. We call memory *accessible* if there is some way for a domain to access it without causing a fault, i.e., by reading, writing, or executing it. Memory is *shared* when it is accessible by more than one domain.

Memory ownership

Memory *ownership* is a component of permissions policy that is implemented entirely within the memory supervisor. Ownership identifies the domain that has ultimate authority on permissions and use of a memory region. The address space is divided into non-overlapping regions, where each region is owned by exactly one protection domain. The supervisor itself owns all of memory initially. An owner can set arbitrary access permissions on memory that it owns, and can grant arbitrary access permissions or export permissions on that memory to other domains. For example, a domain that wants to generate code would give itself read-write permission on the buffer, write the code, then change its permissions to execute-read.

Memory ownership is much coarser grain than memory protection, and changes much less frequently. The supervisor maintains ownership information using a sorted list of memory regions and their owners.

In Mondrix, the only way for a domain to cede ownership of memory is to create a new domain using that memory. The supervisor could provide a `chown` call, which would allow a domain to give ownership of a memory region to another domain, but this was not found necessary.

Export permissions

The memory supervisor also implements *export permissions*, which describe how a domain can grant permissions to another domain. Ownership conveys unlimited export permissions, but non-owner domains can have restricted export permissions. For instance, an owner domain can give another domain (call it domain X) read-write access permissions on a buffer, but limit it to read-only export permissions. Domain X can read and write the buffer, but cannot grant read-write permissions on the buffer to a third domain Y.

The current Mondrix supervisor implements a limited form of export permissions, based on ownership and access permissions. An owner can export permissions freely, while a non-owner can export only up to its access permissions level.

3.2 Permissions and memory allocation

The Mondrix design allows protection, entirely managed by the supervisor, to be separated from dynamic memory allocation, which is managed by the kernel. This allows the main kernel allocators (the page and the slab [5] allocator) to remain outside the supervisor, and lets the kernel retain custom memory allocators, i.e., allocators that manage their own free list such as the Linux inode or socket allocators.

The supervisor provides special API calls (`perm_alloc` and `perm_free`) to support allocators that provide memory to other domains. A domain (call it domain X) calls an allocator domain (call it domain Z), and the allocator domain determines the start

address and length of the memory that X will receive. The allocator domain then calls the memory supervisor to establish permissions for X on the memory it has chosen. The supervisor determines that the permissions are for X by reading the cross-domain call stack.

An allocator domain can own the memory it allocates, which is the fast path used by the slab and page allocators, or it can have export permissions, which is used by the custom allocators because they do not own the memory they allocate; they get it from the slab or page allocator.

The memory supervisor is also responsible for revoking permissions when required, e.g., when a memory region is freed or when a domain is deleted. The supervisor must revoke the permissions because it can not trust other domains to do so correctly.

The supervisor keeps track of which domains have access permissions to memory. This has three significant advantages. The first advantage is that the supervisor prevents domains from leaking permissions by automatically deleting permissions when necessary. The second advantage is that memory need not be tracked by kernel code after it is allocated. The owning domain simply shares the memory, and frees it as usual. The owning domain does not need to track the domains to which it exports permissions, reducing the changes in kernel code to use MMP. The final advantage is that revoking permissions from only the domains that have it is significantly faster than checking all domains for access rights.

3.3 Thread-local stack permissions

The memory supervisor is responsible for managing thread-local stack permissions. Threads can control permissions only for frames in their current domain; the supervisor rejects permission change requests for memory between the most recently saved frame base and the stack base. If a thread grants write permission to a frame, it must revoke permissions on the frame before the frame returns, or it will leak permissions (just as a domain which does not properly revoke permissions on a buffer leaks permissions). On scheduling events, the kernel calls the supervisor to save and reload the thread-local CPU registers (`sb`, `fb`, `sl`, `CDST`, and the stack table base).

3.4 Creating and deleting domains

The supervisor manages the creation and deletion of protection domains. A domain can create a new domain by *subdividing*, passing ownership of a region of its own memory to the new child domain. The supervisor tracks the parental relationships between domains using a tree, with the supervisor itself at the root. When a domain is deleted, ownership of its memory regions passes to its closest extant ancestor. The supervisor must also revoke permissions on memory owned by the deleted domain from all domains.

3.5 Permissions policy

Table 1 summarizes the supervisor's API and policies for managing memory ownership and permissions. There are two calls to set permissions on memory regions: `mprot` sets permission for the current domain while `mprot_export` sets permission in another domain. The `pd_subdivide` call creates a new domain, while `pd_free` deletes a domain. Memory allocator domains call the supervisor `perm_alloc` and `perm_free` routines to give the caller of the allocator access permissions in the memory being allocated.

While there are many details in the table, the supervisor policy follows a few general rules: a non-owner can not dictate permissions to an owner; a non-owner can not downgrade the permissions of another domain; a non-owner can not upgrade its own permissions.

Table 1 refers to an ordering on permissions values. Mondrix uses a partial order. Read-write, execute-read, and gate permissions

Before				Call	After				Comments
Caller		Target			Caller		Target		
own?	access	own?	access		own?	access	own?	access	
y	X			mprot(ptr, len, A);	y	A			An owner can grant itself arbitrary permissions.
n	B			// Change own permissions.	n	A≤B ? A : ERROR			A non-owner can only downgrade its permissions.
y	X	n	Y	mprot_export(ptr, len, C, target);	y	X	n	C	An owner can override a domain's permissions.
n	X	y	Y	// Change others permissions.	n	X	y	ERROR	It is an error for a non-owner to override an owner's permissions.
n	D	n	E		n	C≤D ? D : ERROR	n	C≥E ? C : ERROR	A non-owner can only export at its access level, and can only upgrade another non-owner's permissions.
y	X	n	none	pd_subdivide(ptr, len, E);	n	none	y	E	A domain can only subdivide with memory it owns and does not share.
n	X			// Create new domain.	n	ERROR			A domain cannot subdivide with memory it does not own.
n	X	y	Y	pd_free(target); // Delete domain.	y/n	none			The supervisor revokes permissions on memory owned by a deleted domain from all other domains. The memory owned by the deleted domain becomes owned by its parent, which may or may not have been the caller.
y	X	n	none	perm_alloc(ptr, len);	y	X	n	RW	When the allocator owns memory, it allocates it with read-write permissions.
n	X	y	X	// Access permissions granted to	n	ERROR			A domain cannot allocate memory to the memory's owner.
n	F	n	G	// caller of memory allocator.	n	G	n	F≥G ? F : ERROR	A non-owning domain allocates at the access permission it has, and cannot downgrade the permissions of another non-owning domain.
y	X	n	X	perm_free(ptr, len);	y	X	n	none	A free revokes permission from all sharing domains.
n	X			// Access permissions revoked // from caller of memory allocator	n	ERROR			A non-owning domain cannot free memory.

Table 1: Memory supervisor policy for memory ownership and permissions. The Before column shows the state of the calling domain and the target domain before the supervisor call, identified by the Call column. The After column shows the state after the call. A 'y' (or 'n') in the own? column indicates the domain owns (or does not own) the memory being manipulated. An 'X' or 'Y' in an access column indicates an arbitrary memory access permission, though an 'X' in the before and after columns indicates the value has not changed. Other uppercase letters indicate a specific (but arbitrary) permissions value; "none" indicates no permissions; "RW" indicates read-write permissions. Columns for domains not involved in a particular call are left empty. An ERROR outcome anywhere in a row indicates the supervisor call returns an error for that call. The operator ? :, borrowed from the C language, indicates conditional state.

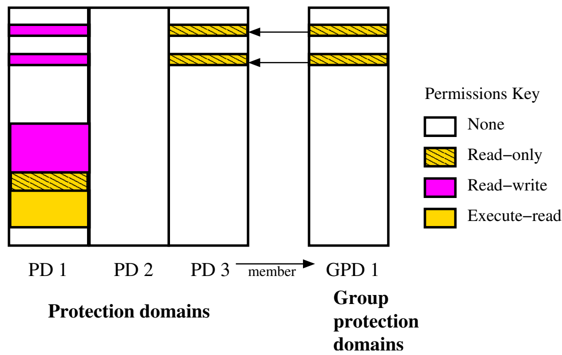

Figure 4: An example of a group protection domain. In this case, protection domain 1 has read-write permissions on two regions of memory. It grants read-only permissions on both to group protection domain 1. Protection domain 3 joins the group protection domain (indicted by the arrow labeled "member"), gaining read-only permission to the two pieces of memory from GPD 1 (indicated by the two arrows from GPD 1 to PD 3).

all compare equal, so a non-owning domain can convert between these permissions values. All of these values compare greater than read-only, which in turn compares greater than no permissions.

3.6 Group protection domains

A *group protection domain* is a collection of memory regions, each with a specified permission. Group domains are useful when multiple domains need access to the same set of memory regions, and where the memory segments in that set change over time. They are not essential to Mondrix's function, but they are a powerful tool.

A regular protection domain can create a group protection domain and then grant access permissions to the group for multiple memory segments. Another protection domain can then *join* the group domain to gain the permissions specified by the segments in the group. This process is shown in Figure 4, where PD 1 exports two memory regions read-only to a group protection domain that is joined by PD 3. When a domain grants or revokes permissions to a group, the memory supervisor adds or revokes permissions on the new memory for every domain in the group. All domains are members of a special global group maintained by the supervisor that contains memory regions with global access permissions. The supervisor can reduce the cost of updating group permissions by sharing appropriately aligned pieces of the underlying trie-based permission table across domains.

One example use of group domains is for the kernel inode structure, which records metadata information for file system objects. Several modules (such as the EIDE disc driver and the interpreter loader) need read access to inodes. The kernel creates a read-only group protection domain of inodes that a module can join to get read permissions on these memory areas. The memory locations that hold inodes change over time as inodes are allocated and deleted, and the kernel keeps the group protection domain of inodes up to date by adding the new ones to the group, and deleting the old ones from the group.

The memory supervisor regulates which protection domains can join a group. Group domains, like any access control mechanism with groups [32], must address difficult issues of how group membership is managed. The memory supervisor would enforce the policy chosen by the system designer, but we defer to the literature for possible policies, and simply present the group mechanism.

4. COMPARTMENTALIZING LINUX

The Mondrix prototype partitions the Linux kernel into protected modules using MMP. This section first describes how Mondrix divides the Linux kernel code into protection domains, and then describes the code Mondrix adds to Linux to explicitly manage memory access permissions, cross-domain calls, and interrupts.

4.1 Mondrix module structure

Module	Description
mem. supervisor (bot)	Code that writes the MMP permissions tables.
mem. supervisor (top)	Code to manage device-independent MMP permission abstractions.
kernel	Most of the Linux operating system.
printk	This is an ad-hoc collection of the kernel functions and data consisting of `printk` and related functions (e.g., `sprintf`, vsprintf).
ide-mod ide-disk ide-probe-mod	Collectively, the EIDE disk driver.
unix	Unix domain sockets (used by syslogd).
rtc	The real time clock.
binfmt_misc	The interpreter loader (supporting #!/bin/sh).
8390 ne	The bottom and top halves of the network driver, controlling an NE2000 network interface card.

Table 2: Mondrix kernel modules. Each module is resident in its own domain even when several modules share a description.

The code in Mondrix is divided into the protection domains shown in Table 2. The partition of code into domains is arbitrary, but Mondrix uses several guiding principles. It isolates each kernel module in its own domain. If the kernel developers think of a collection of functions as a module, then that collection can control its memory permissions in Mondrix. The disk and network device drivers are sub-divided into several modules and each module resides in its own domain.

Mondrix also collects certain functions into domains in order to increase memory isolation within the kernel. Domain 0 holds the bottom half of the memory supervisor. Domain 1 holds the top half of the memory supervisor. Most of the kernel is resident in domain 2, while domain 3 holds the collection of kernel functions that print, write and format strings (included the dreaded `sprintf` function, cause of many buffer overflows).

The division into protection domains forces all memory sharing between modules to be explicit. The principle of least privilege dictates that each kernel module has the minimum memory permissions necessary for correct operation, but this desire must be balanced against performance and ease of programming. For example, the `printk` domain was granted permission to read all kernel strings. Kernel strings are contiguous in memory and granting read permission for each string would be tedious, error-prone and less efficient. Individual modules export read permission to `printk` for individual strings or for their string section. All write access to stack variables (mostly for `proc` filesystem calls to `sprintf`) is provided and removed explicitly.

4.2 Loading modules into protection domains

Linux kernel modules are object files that a user loads into a running kernel using the `insmod` program. The `insmod` program reads a module from disk, then links it against the currently running

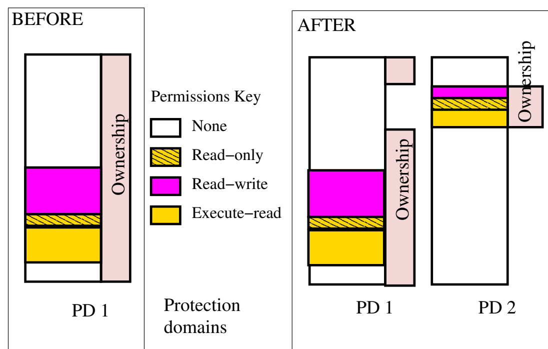

Figure 5: A before and after picture for domain creation with module loading. For each domain, the thicker bar shows the protection information, and the thinner side bar shows ownership information.

kernel (based on symbol information it receives from a system call), resolving any undefined symbols in the object module.

After checking the module, the kernel calls the memory supervisor to set correct memory permissions on the module. The supervisor needs the length of the program sections (already provided by `insmod`), and for every function its start address (also already provided) and the address of the return instruction (Mondrix's `insmod` provides this additional information). Program section information is used to properly set the initial permissions for the module, while function entry and exit information is used to guarantee that switch and return gates are set only at the start and at the return instruction of a function respectively. The memory supervisor places the obvious permissions on each section (e.g., execute-read permission on the text section).

The supervisor needs the address of the return instruction for public functions in the module that other domains call, so it can set a return gate on the function. With `gcc` versions 3.3.x, the return instruction is placed arbitrarily in the function to allow outlining of uncommonly executed code, so Mondrix stores the return instruction addresses in the kernel modules.

Figure 5 shows how permissions and ownership information change when a domain is created to hold a newly loaded module. In the before state, the kernel (in PD 1) owns all of physical memory. In the after state, it has subdivided, and loaded a module into PD 2. Permissions for the module's code and static data are given by the shaded regions, and correspond to the object file layout of program sections. The kernel allows the module in PD 2 to own its static code and data, but it retains ownership of the rest of the address space.

Previous work [42] predicted that symbol information could implicitly define sharing relationships. Code and data can be exported by name (using the `EXPORT_SYMBOL` directive in Linux). While this is the currently encouraged method for exporting functions in the Linux kernel, there is plenty of legacy code that exports functions anonymously by passing their address directly. Kernel code assigns the addresses of regular and static functions to structures of function pointers that it passes to other domains. The names of the static functions are not visible to the other domains, but the function pointers are! Unfortunately, import and export of function name symbols is only an incomplete record of true inter-domain calling behavior.

As bad as the situation is for code, it is worse for data. C's ambiguity between pointer and array, and the relative rarity of importing data by named symbol makes imports nearly useless as an indicator of true inter-domain data sharing. Most code uses `extern` declarations for data instead of `EXPORT_SYMBOL`.

4.3 Disk driver

Many kernel drivers are split into two parts, a device-dependent bottom half and a device-independent top half, where each half is an independently loaded kernel module. The EIDE disk driver has one top half (`ide-mod`) and two bottom halves, one to gather disk controller information (`ide-probe-mod`), and one to gather disk geometry information (`ide-disk`). Different halves of a driver share data structures, and call each other frequently.

Adapting device drivers for Mondrix consists of placing explicit calls to the memory supervisor that manage access permissions on memory used by the driver and shared with other parts of the system. In order to increase memory isolation, Mondrix grants permission on page and buffer cache memory regions before calling the disk driver to read or write the data, and revokes permissions once the I/O is done. The EIDE disk model in Bochs does not support DMA, but the EIDE disk model in SimICS does, so Mondrix controls the permissions for both DMA and programmed I/O data transfer.

Another part of device safety is proper programming of device registers. In one file system corruption prevention experiment (Section 5.1.2), Mondrix was able to determine that the disk controller was programmed with a bad address range because it read from an illegal location. Mondrix could do more of this kind of checking; specifically it could check memory bounds for every I/O request (DMA or PIO) without changing the interface. A small I/O bounds check domain is created, and only this domain gets write access to memory mapped I/O device control registers. The kernel or driver calls into the I/O domain to write to the DMA engine registers. The bounds checker checks the values and performs the writes. Writing device control registers is already slow compared to normal memory references, so the additional latency of the cross-domain call and check should not have a significant impact on overall performance. This approach would catch DMA programming errors, but would not prevent a faulty device from writing out of its programmed bounds.

4.4 Network driver

The NE2000 network driver has a chip-specific portion (`8390`), which coordinates the reception and transmission of packets and handles device interrupts and initialization, and a board-specific portion (`ne`), which moves data onto and off of the network card. Mondrix must give the `8390` module read-write permissions on certain fields in the `sk_buff`, which is the kernel data structure which manages packet data. It must give the `ne` module read or read-write permission (for transmission or reception) on the packet data itself.

Mondrix allows the `ne` module to retain read permission on packet data while packets can be retransmitted. It allows read-write access to the `8390` module to 8 words (32 of 144 bytes) in the `sk_buff` structure, none of which are kernel data structure pointers (though some of the fields point to packet data). This policy limits the damage a malfunctioning driver can do to the kernel and increases the chances that a malfunctioning driver would be detected by an illegal memory reference. Mondrix allows kernel programmers to balance memory isolation with performance. A more restrictive permissions policy would remove write access to the `8390` module for `sk_buffs` that are on the free list.

It is unfortunate that the only Ethernet device model Bochs supports is the NE2000, since the NE2000 is not a sophisticated device (it does not support DMA), but adding an additional Ethernet hard-

ware model to the Bochs machine simulator was beyond the scope of this work.

4.5 Interrupts

Handling device interrupts is an important operating system task, and MMP allows them to proceed in a protected way. Interrupts do not cause a protection domain switch, but jump to shared interrupt stubs that are marked executable in every domain using the global group protection domain. The interrupt assembly stubs are a shared library, albeit a simple one that has no data.

The stubs must be verified by inspection, as they are now (about 50 lines of assembly code), since they are trusted in every domain. The transfer from the interrupt stub to a C handler routine has a switch gate, causing a domain crossing to the handler's domain. Distributing the assembly stubs to all protection domains does not create a new vulnerability since the correct functioning of the machine is dependent on the correct functioning of the interrupt assembly stubs.

4.6 Inlining

In C, header files sometimes include inlined functions that reference a module's internal data. Any domain that calls the inlined function needs permission to access the inlined data. Sometimes the domain exporting the inlined function should export permissions on its data, and sometimes an inlined function should be uninlined to avoid giving other domains permission to read or write its sensitive data. Mondrix uses both approaches, on a case by case basis.

4.7 Slab allocator

The kernel slab allocator [5] is called frequently for small memory objects, allocated out of caches (`kmem_cache_t`). Mondrix takes advantage of the fact that the domain that allocates a cache is almost always the one that allocates memory from it. Mondrix manages the permissions for entire slabs (usually pages) internal to the caches. This does not compromise safety because the supervisor checks (with hardware providing integrity) if the calling domain owns the cache, and if not, provides permissions only for the individual object requested. However with this policy, a domain that owns a cache can write into memory that was not yet allocated and not cause a fault.

Like many decisions about how tightly to control memory permissions in Mondrix, optimizing the slab allocator trades speed and isolation. Mondrix chooses speed in this case, but the fault injection results (Section 5.1.2) indicate a high degree of isolation with this policy.

5. EXPERIMENTAL EVALUATION

This section analyzes the performance of Mondrix executing on the SimICS [27] and Bochs [23] machine simulator. We added a functional model of the MMP hardware to each simulator, and booted Mondrix on the modified simulator. The memory supervisor in Mondrix handles all permissions requests, and its bottom half writes the permissions tables, so all instruction and memory traffic from that code is present. The model includes a cache simulation, gathers workload statistics, and checks all accesses for correct permissions.

5.1 Functional evaluation

Our hypothesis when building Mondrix was that the memory isolation it provides would allow the kernel to detect data structure corruption, limiting damage from bugs. Mondrix exposed a latent bug in Linux, and we injected faults into Mondrix to see how effective it would be at detecting and avoiding data structure corruption.

5.1.1 Mondrix exposes a Linux error

Converting Linux to Mondrix exposed a case where, during kernel initialization, the kernel freed the stack memory on which it was executing. The kernel continued to use the stack memory after it freed it, even making calls into dynamically loaded modules.

`proc_pid_lookup` is a function in the proc file system (a pseudo-filesystem for processes control and information) that looks up a user area based on the process identifier. The function calls `free_task_struct` on the task it looks up. The call should not actually free the task structure because the function decrements a reference count that was incremented earlier in `proc_pid_lookup`. `free_task_struct` only frees the task structure if the structure's reference count is zero. But the reference count is zero at one point during kernel initialization, so `free_task_struct` actually frees the task structure. Since the task structure and the kernel stack are in the same allocation unit, the kernel stack is freed along with the task structure. In one case, the kernel frees the memory for the stack on which it is executing. Since the Mondrix memory supervisor revokes all permissions on memory that is freed, it reports many protection violations from the kernel reading and writing the stack memory it just freed.

Another call to `free_task_struct` is made in `proc_pid_delete_inode`, where it should be balanced by a previous increment of the use count on the task_struct memory. But again this routine causes the kernel to free the stack memory on which it is executing. The code that manipulates the reference counts for the task structure was changed during the development of version 2.5, and versions 2.6.x use the new system. We did not check if the new code manifests the same bug we found in 2.4.19, because we have not ported our kernel changes to 2.6.x.

5.1.2 Fault injection experiments

In order to demonstrate Mondrix's effectiveness at containing memory corruption in the presence of kernel bugs, we ran a series of experiments injecting faults into Mondrix. We use the same fault injection code used in the Rio file cache studies [8] and Nooks [38], which changes instructions and data in the kernel binary in a way that models the effect of real software bugs. Once the kernel loads all of its modules, the fault injection code injects faults and the kernel tries to run a small workload consisting of `find` and `wget` to simulate normal use of the disk and network.

Corrupting the file system is one of the worst possible outcomes from a kernel failure. After each fault injection experiment we ran the Unix file system consistency check program `fsck`. If `fsck` deleted files or directories in its effort to reconstruct the file system, we classified that run as corrupting the file system. Deleting files and directories goes beyond the metadata fixups (e.g., fixing the free block count) that are common from a kernel crash or hang.

These experiments were run on Linux's `ext2` filesystem. There are journaling file systems (like `ext3`) that largely avoid the problem of corrupted file systems due to unexpected crashes or kernel behavior. The purpose of the experiment is to show that Mondrix can catch the effect of kernel problems before they propagate and spread to other parts of the kernel. A corrupt file system is a reasonably common and unpleasant example of kernel bugs' effects rippling out from their point of origin.

Five out of 200 fault injection experiments resulted in a corrupt file system. In three of these cases, Mondrix detected a memory permission violation, and in all three cases if Mondrix halted when the MMP protection system detected the violation, it would not

Symptom	# runs	MMP catch
None	157	4 (02.5%)
Hang	23	9 (39.1%)
Panic	20	18 (90.0%)

Table 3: Fault injection experiments on Mondrix. Faults either resulted in a clean shutdown (this category includes cases where the faulting process (and/or others) is terminated), a hang or a panic. The # runs column shows the number of instances for each symptom (200 runs total). The MMP catch column indicates the number of runs where Mondrix caught a memory permissions problem (which was not caught by the kernel's page table).

have corrupted the file system.

The three cases are interesting because they display the strengths of MMP, and the diversity of kernel failure symptoms. In one case (simulating a pointer dereference bug) MMP catches the EIDE disk controller reading from dynamically allocated kernel memory. The disk controller does not have access to that memory, but it was passed a bad pointer. In another case (in which a random instruction was deleted), `sprintf` was passed a pointer to a device lock instead of a character buffer, and it corrupted the lock and nearby data structures. In the final case (simulating failure to initialize a variable), the console driver reads from an address that could be the address of a kernel stack, but is not.

The proper strategy for dealing with faults in the kernel depends on how the operator wants to balance availability with data integrity. Linux's default behavior on a kernel memory fault is to kill the process context that caused the fault. This can be an effective way of limiting the scope of the problem while keeping the system running. For instance, in one of our fault injection experiments the kernel killed `modprobe` after it finished loading a module. Because the user process had completed its work, killing the context was a safe and effective course of action. Some faults are more serious, resulting in hangs or kernel panics. Mondrix can detect when faults are corrupting data structures and stop them to limit the scope of the damage.

The data in Table 3 summarize our fault injection experiments. MMP detected more illegal memory sharing as the symptoms of a fault rose in severity from minor symptoms (possibly killing a user task) to system hangs and kernel panics. This data is suggestive that MMP is detecting important errors as illegal memory sharing, especially in conjunction with the data about file system corruption. However, a lack of symptoms does not always imply correct operation, nor does a hang necessarily imply a major problem. Important data structures can be corrupted when the kernel successfully shuts down, and some hangs occur late in shutdown where they are benign. Subdividing the kernel into more domains might catch more memory data structure corruption (and cost more performance).

5.2 Performance evaluation

We use a performance model to estimate the overhead of adding fine-grained memory protection support to the processor. We assume a processor that can complete one x86 instruction per cycle. We model a two level cache hierarchy, based on the Intel Pentium-4, with 16 KB 4-way associative level-one instruction and data caches, and an 8-way associative 2 MB level-two unified cache. Level-one miss penalty is 16 cycles, and level-two miss penalty is 200 cycles (this memory access penalty is low, representing a 4 GHz processor able to access local DRAM in 50 ns). Cache lines are 64 bytes. Main memory size is 256 MB, which is small by today's standards, but is a limitation of the simulation environment. This model represents the performance of an aggressive processor over the next few years.

Benchmark	Description
config-xemacs	./configure for xemacs 21.4.14
thttpd	A small http server (thttpd) serves 452 KB of data from 28 requests, 13 of which require forking a cgi script which run several programs.
find	find /usr -print | xargs grep kangaroo; /usr is 255 MB, 1,720 directories with 16,343 files.
MySQL	A MySQL client test from the MySQL distribution. The client connects to the database (on the same machine) and executes 150 test transactions covering the range of database functionality.

Table 4: The names and descriptions of the benchmarks run on Mondrix to evaluate MMP support in the Linux kernel.

SimICS EIDE disk model properly limits disk bandwidth and provides a simple fixed latency for each disk operation. We use a disk latency of 5.5 ms, representing an aggressive 2 ms for the rotational latency of a 15K RPM disk and an average seek time of 3.3 ms (the disk-active workloads make random requests making this number optimistic). The SimICS EIDE disk model includes DMA.

Table 4 shows the system-intensive benchmarks we ran on Mondrix to measure the effect of isolating kernel modules in separate protection domains. The benchmarks were chosen as common tasks that stress the disk and network subsystems of Mondrix. The OS was booted fresh before each trial. All utilities were from the Debian Linux distribution as of January, 2005.

The configuration of `xemacs` is a long running test that stresses the virtual memory system with process creations, deletions, scheduling and small file access. It runs for long enough that the kernel memory allocators reclaim memory. The `thttpd` benchmark is a small web server that serves data and runs cgi scripts. The cgi scripts in turn run several native programs e.g., to print environment variables. This benchmark uses the network heavily and also creates many small processes. The `find` benchmark is disk and filesystem intensive, as is the `MySQL` database test.

The graphs in Figure 6 (best viewed in color) show the performance of the benchmarks on Mondrix. `config-xemacs`, `find` and `MySQL` run on SimICS, while `thttpd` runs on Bochs (because SimICS does not have an NE2000 network device model). SimICS's EIDE disk model supports DMA, so the workloads run on SimICS spend only a small amount of time servicing disk interrupts. The NE2000 network device in Bochs does not use DMA, so the time to service network interrupts includes the time for the processor to copy the packet data. Most of the kernel `other` category in the `thttpd` workload includes data copying from interrupt processing. Bochs does not model the device latency of the network card.

The CPU overhead of adding MMP to Mondrix is less than 15% for all benchmarks, and below 8% for the non-networking benchmarks. As explained in Section 4.4, Mondrix tightly controls the permissions on network packets. The numerous calls to `mprot_export` from the kernel's networking code shows up as time spent writing permissions tables (`mmp_bot` in the graph). The MMP overhead could be reduced further by using a pool of pre-

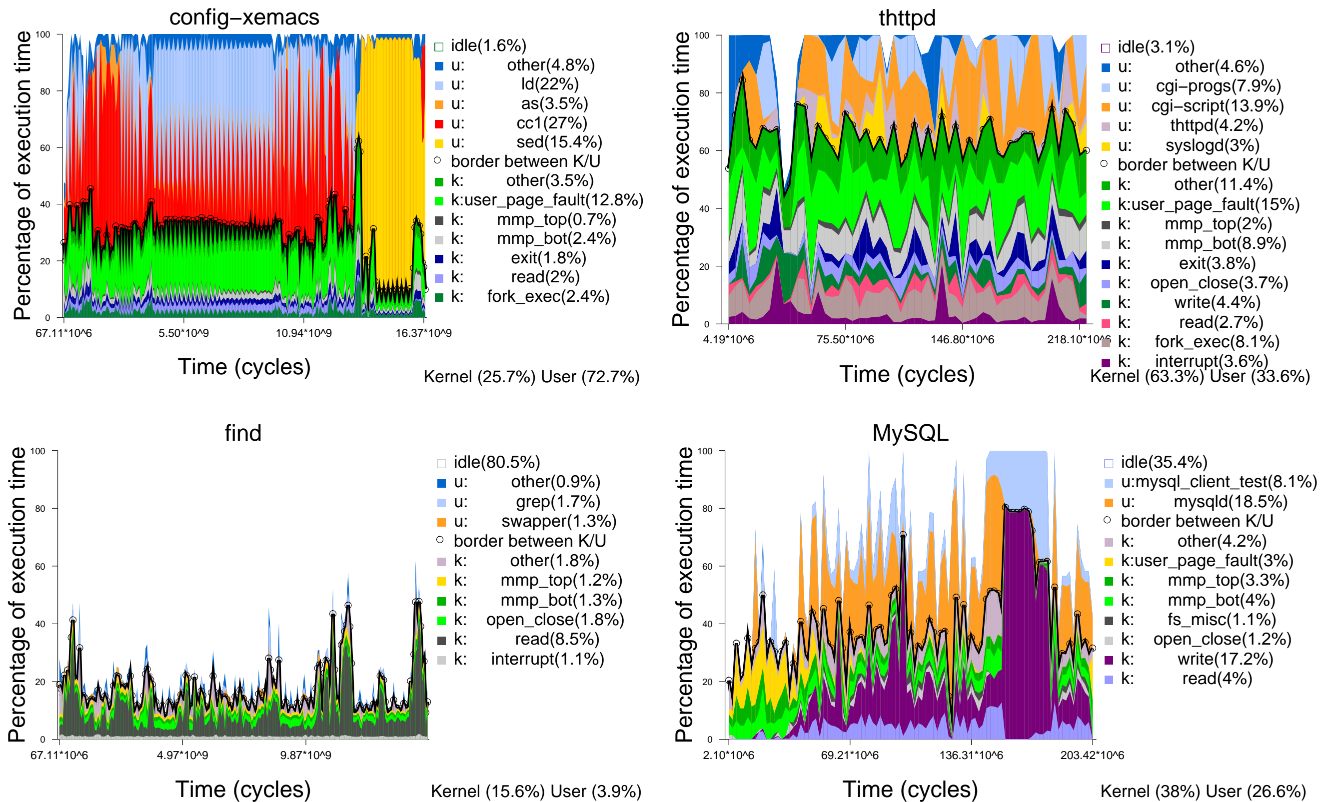

Figure 6: Performance of benchmarks on Mondrix, including instructions, memory stalls, and disk device latency. The `other` category for user programs is for any program whose individual contribution to performance falls below a threshold. The `other` category for the kernel includes system calls, kernel threads and interrupt processing that falls below a threshold. Categories like `fork_exec` include all system calls related to forking and execing processes, and `fs_misc` includes most file system calls that are not `open`, `close`, `read`, or `write`. The `mmp_bot` category is the bottom of the Mondrix memory supervisor that writes the protection tables, while `mmp_top` is the top half of the memory supervisor. The workload's kernel/user execution time split appears at the bottom of the legend.

allocated packet buffers.

Table 5 shows the performance overhead of Mondrix as compared with an unmodified Linux. CPU and memory overhead is less than 15%. All experiments described in this section use the bitvector format [43] for the permissions tables.

Permissions tables are written in response to memory allocation, process creation, and direct calls to the memory supervisor. The slab allocator optimizations (see Section 4.7) are effective at limiting table updates due to memory allocation. The `config-xemacs` and `thttpd` benchmark create many processes, and see increased table writing activity because of it. The kernel could keep tables for a process' program sections resident while most of the process' text pages are resident, reducing the overhead of re-executing the same process. Direct calls to the memory supervisor are a matter of programmer policy. Mondrix's tight control of permissions on network buffers is the main contributor to the performance overhead in the bottom of the memory supervisor. Much of the kernel overhead for `thttpd` arises from increased memory traffic due to PLB refills, as explained in Section 5.2.3.

Benchmark	Cyc($\cdot 10^9$)		Mbot	Mtop	Kern
config-xemacs	16.5	(4.4%)	2.4%	0.7%	1.3%
thttpd	0.23	(14.8%)	9.3%	2.0%	3.7%
find	14.3	(3.3%)	1.3%	1.2%	0.8%
MySQL	0.21	(9.6%)	4.0%	3.3%	2.3%

Table 5: Performance overheads for workloads on Mondrix, as compared with Linux. The Cyc column shows the number of cycles (in billions) for the workloads, and in parenthesis, the slowdown of the workload compared with Linux. The Mbot column shows the percentage of time spent in the bottom half of the memory supervisor, writing permissions tables, while Mtop shows the time spent in the top half of the memory supervisor. The Kern column shows the overhead in the remaining kernel code including code added to the kernel to manage memory permissions and PLB refills.

To keep the overhead from the bottom of the memory supervisor low, the table writing code is heavily optimized. The table writing code uses lookup tables to write permissions in 32-bit words, and the code is optimized to quickly find the proper table given that most allocations are for a page or less.

The `find` benchmark and `MySQL` make heavy use of the file system, creating significant idle time. This idle time overlaps much of the Mondrix overhead caused by the additional checks in the inode allocator, the generic block driver, and the slab memory allocator. The code in these subsystems calls the memory supervisor and makes local decisions about granting memory access.

Benchmark	XD	Ca	Cy/Ca	Self/Other	
config-xemacs	0.3%	3.29	1,286	70%	30%
thttpd	0.8%	0.15	939	64%	36%
find	0.2%	2.74	846	57%	43%
MySQL	0.7%	0.12	664	50%	50%

Table 6: Cross-domain calling behavior for workloads running on Mondrix. The XD column is the percentage of total execution time each workload spends doing cross-domain calls (including compute cycles and memory references). The Ca column is the number of cross-domain calls in millions. The Cy/Ca column is the average number of non-idle kernel cycles (instruction and memory stall) between cross-domain calls. The Self/Other column indicates the percentage of cross-domain calls that a domain makes to itself versus those that cause a domain change.

There were several challenges running the network experiments on Bochs. There is a bug in the Bochs device model which causes occasional transmit errors at the device level. The occurrence of these bugs can be seen as the idle time in the graph as the kernel resets the network card and retransmits the lost packets. Thirteen packet corruptions occurred in this run out of 498 packets (eight packet corruptions occurred in the corresponding test on unmodified Linux).

In order to minimize the timeout bug and because the simulator does more work (checking permissions) when running Mondrix, simulated time runs at different rates in the Linux and Mondrix benchmarks. Under unmodified Linux the system believes that 40 seconds have elapsed, while under Mondrix the system believes that only 2 seconds have elapsed. The amount of web server related work is the same, but under Linux the benchmark does more user work (`klogd` runs under Linux and does not have time to run under Mondrix). To compensate we compare only webserver related time for `thttpd`. The overheads from the memory supervisor are higher in Table 5 than in Figure 6 because the idle time was subtracted from the total runtime (normalizing to webserver related work) to compute the figures in the table. Only `thttpd` runs under Bochs, so it is the only benchmark with this problem.

We investigated removing permissions from `sk_buffs` when Mondrix places them on the free list and reinstating the permissions when they are dequeued. That brings the total overhead up to 19.1% from the 14.8% in the table.

5.2.1 Evaluation of cross-domain calling in Mondrix

Table 6 summarizes cross-domain calls in Mondrix. Cross-domain calls account for less than 1% of the total execution time for all benchmarks. The protection domain granularity enforced in Mondrix is very fine-grained (justifying architectural support). The table shows that cross-domain calls are frequent (at least once every thousand cycles of kernel activity), and cross-domain calls from a domain to itself are more frequent than calls to other domains. A domain makes cross-domain calls to itself when it calls a function that it also exports to another domain (like the kernel exports `kmalloc`). As the kernel is split into more domains, more calls will be cross-domain calls.

The cost model for a cross-domain call or return is a 5 cycle penalty to flush the pipeline and perform memory accesses plus any memory stall accrued by the memory accesses. Each cross-domain call stores the protection domain, return address and if the protection domain changes, the contents of the `fb` register. Each cross-domain return loads these values (only loading `fb` if the protection domain changes). The loads for the permission table base pointers are not included since these can be cached on chip. During all of these benchmarks, the cross-domain call stack never grows deeper than 64 entries, so this data structure does not occupy significant cache area.

Benchmark	Free Mem Used
config-xemacs	10.2%
thttpd	1.1%
find	7.8%
MySQL	1.6%

Table 7: Reduction in free kernel memory after each workload as reported by /proc/meminfo.

5.2.2 Evaluation of memory use in Mondrix

Table 7 shows the memory overhead of Mondrix by comparing the output of /proc/meminfo after each benchmark for Mondrix and an unmodified Linux. The memory overhead represents how much less free memory the kernel has after running each benchmark because that is the most conservative metric. For all benchmarks the sum of the `Active` and `Inactive` memory in the kernel was within 1% for Mondrix and Linux. The memory supervisor's data structures do not disturb the kernel's active memory use.

5.2.3 PLB refill traffic

Table 8 shows how effective the on-chip protections cache (the protection lookaside buffer (PLB)) is at caching permissions. On a PLB miss, the cost model for the refill is 1 cycle per load plus any memory stall the load incurs. The PLB caches permissions data for heap, text and stack memory (but it does not cache gate information). All benchmarks spend less than 4% of their execution time refilling the PLB. The PLB refill cost is spread through execution of all domains including the memory supervisor. The networking benchmark writes the permissions table frequently, so it spends the most time refilling the PLB.

The memory supervisor keeps the PLB consistent with the permissions table by flushing the PLB when necessary. The PLB is implemented as a ternary CAM [43], so permissions can be flushed for power-of-two sized virtual address regions. The supervisor flushes the PLB for address ranges that could become stale when it writes the protection tables, and it flushes the PLB of stack permissions on a process switch.

The PLB miss rate is a bit lower than typical second level cache miss rates. The PLB can effectively cache information for regions larger than a page, like the kernel text and data sections.

6. RELATED WORK

Nooks [37, 38] provides device driver safety using conventional hardware. Nooks uses conventional paging hardware to isolate modules by putting them in different addressing contexts (protec-

Benchmark	PLB time	PLB mr
config-xemacs	0.8%	0.51%
thttpd	3.8%	0.87%
find	0.4%	0.07%
MySQL	1.7%	0.22%

Table 8: The `PLB time` column is the percentage execution time of each benchmark that the hardware refills the PLB. The `PLB mr` column is the miss rate of the PLB.

tion domains). These domains execute with full kernel privileges, but they differ in their view of memory permissions. Crossing Nook boundaries is expensive because it requires changing virtual address context and copying parameters. To minimize boundary crossings, Nooks places multiple kernel modules in the same protection domain. MMP can enforce the natural, fine-grained module boundaries established by the Linux kernel developers. The frequency of cross-domain calls in the MMP system (Section 5.2.1) is at least an order of magnitude greater than Nooks [37] without a decrease in performance, indicating that MMP offers greater modularity and isolation.

Nooks is an elegant solution to the specific problem of bringing safety to OS extensions for existing hardware, while MMP is a proposal for a general-purpose architectural mechanism for protected sharing, which we have applied to the problem of safe OS extensions in this paper. MMP can also be used to provide safe user extensions, and a variety of other applications like data watchpoints, optimistic compiler optimizations, and efficient read barriers for garbage collection.

Mondrix contains modules that are not device drivers (like support for unix domain sockets) and whose memory access and calling relationship to the rest of the kernel is not as well behaved as device drivers. It provides protection domains for these ad-hoc modules just as it provides protection for device drivers. Nooks relies on the specific calling relationship drivers have with the kernel and could not isolate modules like the unix domain socket module. In several fault injection experiments where Mondrix caught a sharing violation in advance of a kernel panic, the unix domain was the source of the violation.

Nooks includes a recovery system [38] that can safely restart a failed device driver. It tracks kernel objects and tries to reclaim resources on a fault. Mondrix does not have a recovery mechanism but recovery can be done at a coarser level of granularity than isolation, so Mondrix could use many of Nook's techniques while the MMP hardware should increase the efficiency of the Nooks implementation.

Nooks consists of 22,266 non-comment lines of code, including 924 lines of Linux kernel changes. The top of the Mondrix memory supervisor is 3,922 lines of non-comment code, the bottom is 1,730 and Mondrix requires 1,909 lines of kernel changes. Mondrix requires more kernel changes because memory permissions are managed at a finer granularity, requiring more calls to the memory supervisor. The advantage to adding hardware is that the trusted computing base can be kept small and understandable, as evidenced by the size of the memory supervisor. The functional complexity of the hardware design is quite low, as it has a well-defined behavior that only needs to consider a single dynamic instruction at a time.

6.1 Language-based protection

Microsoft plans to use safe languages (called "managed code") to implement new features in its next next generation operating system, called "Vista"[11]. One of the reasons for the switch to managed code is to provide safety for kernel extensions. Vista's trusted computing base will be orders of magnitude larger than Mondrix's. At this point it is unclear if malicious attacks will be stopped by safe languages, or if attacks will cause resource exhaustion rather than crashes. It is also unclear if the performance cost for the safety of such a system will be acceptable. The switch to a system consisting entirely of managed code (if it can be done) will take many years, during which the vulnerabilities in non-managed code will persist. The operating system and its drivers are a large program to recode, but there is research on how to recode an OS for a safe language [16, 19].

There have been several operating systems that use safe languages as their primary extensibility mechanism [20, 41], with SPIN [4] a large, recent example. SPIN demonstrates how an operating system written in a safe language (Modula-3) can be made efficient in terms of CPU and memory consumption. But device drivers in SPIN are written in C, because rewriting existing driver code is too much work. Also, because of their low-level nature, many device drivers require unsafe programming language features [4]. One advantage of MMP is that it efficiently supports legacy code, written in unsafe languages.

The SPIN project included a linker design [36] whose goals are similar to the gate design in Mondrix. In SPIN, a typesafe reference to a domain gives permission to call that domain's functions. Mondrix allows more fine-grained control. A module may export a different arbitrary subset of its functions to each other domain.

CCured [29, 9] is a language-based approach to adding memory safety to C. It is unclear whether it is more programmer effort to create Mondrix, or to port Linux to CCured. One issue with CCured for operating systems is the requirement of wrappers for libraries not compiled with a CCured compiler. Since proprietary device drivers are often distributed in binary-only format, manufacturers would have to provide wrappers, or they could be reverse engineered. CCured performance is variable, with slowdowns from 0–81%.

6.2 Hardware-based protection

Intel and AMD have announced support for the NX bit in the page table[2], indicating their willingness to add hardware to make software more reliable. Any attempt to execute an instruction from a region with the NX bit set would cause a fault. Security-conscious applications can set the NX bit on their stack, heap, and data sections, that would prevent some malicious attacks. However many attacks overwrite data in jump tables and function pointers, and these attacks will not be prevented by the NX bit.

The operating system for the Cambridge CAP computer [40], and Multics [31] were written to run on hardware that supported capabilities, which provided some of the isolation guarantees of MMP. However, the structure of these systems is different from a modern OS due to hardware imposed restrictions. For instance, Multics limits the number of subsystems in a process to 8, and only allows a subsystem to call another with a higher identifier.

MMP has been compared to segments [24] and capabilities [43]. It has the flexibility of segments, but with the simplicity and backwards compatibility of linear addressing. It provides some of the most attractive features of capabilities, like fine-grained protection domains and flexible resource sharing, while maintaining a backwards compatible programming model and providing simple rights revocation. Hardware capabilities require a fundamentally incompatible change in the processor instruction set, complicate permission revocation [18, 6], and have trouble allowing domains to see different permissions on a region of memory accessed via a shared capability. Many of these problems have been addressed in recent software-based capability systems [34, 35], but the incompatible programming model problem remains a significant hurdle.

XOM [26, 25] is a hardware design and OS implementation of an untrusted OS on trusted hardware. Its goals are different from Mondrix (Mondrix empowers the developer while XOM empowers content providers), but the hardware/OS co-design issues are a close match.

6.3 OS structure

Single-address space operating systems place all processes in a single, large address space [7, 17, 14], and many use protection

domains to specify memory permissions for different thread contexts [21]. The granularity of protection in these systems is a page to match the underlying paging hardware. MMP's finer granularity allows the protection techniques of single address space OSes to be applied to legacy operating systems.

The modularity of Mondrix resembles that found in many microkernel designs [1, 15], but without the performance problems of protection domain switches being coupled with address space switches.

Lightweight virtual machines like Xen [12] and Denali [39] can get some benefits of fault containment by replicating entire OS/application environments. But they do not address the detection of faults within an OS or application, they just provide an alternative to a crash should such an error take place.

7. CONCLUSION

MMP provides a practical solution to the longstanding goal of fine-grained memory protection. MMP provides fine-grained protection with backwards compatibility for operating systems, ISAs and programming models, using only a small amount of additional hardware that is not on the processor critical path. MMP avoids additional confusing programmer-visible abstractions, yet can support most of the best ideas previously proposed for segmented or capability systems.

Our experience in building Mondrix indicates that MMP's programming model fits naturally with how modern software is designed and written. MMP provides hardware enforcement of existing module boundaries, improving software maintainability and robustness. Mondrix's use of hardware memory protection increases Linux's robustness from software errors.

Modularity is a proven technique for providing flexible and stable systems, but current hardware and operating systems provide only crude, and therefore neglected, support for modular software systems. We believe fine-grained memory protection of the kind provided by MMP should be a standard component of future computing platforms.

8. ACKNOWLEDGEMENTS

Thanks to Jungwoo Ha for help with the networking experiments. Thanks to Fletcher Mattox for late night disabling of network port security policies.

9. REFERENCES

[1] M. J. Accetta, R. V. Baron, W. Bolosky, D.B. Golub, R. F. Rashid, A. Tevanian, and M.W. Young. Mach: A new kernel foundation for unix development. In *Proceedings of Summer Usenix*, 1986.

[2] Advanced Micro Devices. `http://www.amd.com/`, 2004.

[3] Thomas Ball and Sriram K. Rajamani. The slam project: Debugging system software via static analysis. In *POPL '02*, 2002.

[4] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan J. Eggers. Extensibility, safety and performance in the SPIN operating system. In *SOSP-15*, pages 267–284, Copper Mountain, Colorado, 1995.

[5] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[6] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *ASPLOS-VI*, pages 319–327, San Jose, California, 1994.

[7] Jeffrey Chase. *An Operating System Structure for Wide-Address Architectures*. PhD thesis, University of Washington, August 1995.

[8] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio file cache: Surviving operating system crashes. In *ASPLOS-VII*, 1996.

[9] J. Condit, M. Harren, S. McPeak, G. Necula, and W. Weimer. CCured in the real world. In *PLDI*, 2003.

[10] Intel Corp. *Intel Itanium Architecture Software Developer's Manual v2.1*, 2002.

[11] Microsoft Corporation. *Microsoft Windows Vista Developer Center*, 2005. `http://msdn.microsoft.com/windowsvista/default.aspx`.

[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. In *SOSP '03*, 2003.

[13] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP-19*, 2003.

[14] Steven M. Hand. Self-paging in the nemesis operating system. In *Operating Systems Design and Implementation*, pages 73–86, 1999.

[15] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schonberg, and Jean Wolter. The performance of microkernel-based systems. In *SOSP-16*, Oct. 1997.

[16] John Hartman, Larry Peterson, Andy Bavier, Peter Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Todd Proebsting, and Oliver Spatscheck. Experiences building a communication-oriented javaos. *Software: Practice and Experience*, 30(10):1107–1126, 2000.

[17] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software Practice and Experience*, 28(9):901–928, 1998.

[18] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. IBM System/38 support for capability-based addressing. In *Proceedings of the 8th Symposium on Computer Architecture*, pages 341–348, May 1981.

[19] Galen Hunt, James Larus, David Tarditi, and Ted Wobber. Broad new os research: Challenges and opportunities. In *Proceedings of the 10th Workshop on Hot Topics in Operation Systems*, June 2005.

[20] Richard K. Johnsson and John D. Wick. An overview of the mesa processor architecture. In *Proceedings of the first international symposium on architectural support for programming languages and operating systems*, 1982.

[21] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architectural support for single address space operating systems. *SIGPLAN Notices*, 27(9):175–186, 1992.

[22] Butler Lampson. Protection. In *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, Princeton University, 1971.

[23] Kevin Lawton. *bochs: The cross platform IA-32 emulator*, 2004. `http://bochs.sourceforge.net/`.

[24] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

[25] David Lie, Chandramohan Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP '03*, 2003.

[26] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick

Lincoln, Ban Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS-IX*, 2000.

[27] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[28] Madanlal Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *OSDI-5*, December 2002.

[29] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[30] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. *ACM SIGPLAN Notices*, 35(5):285–298, 2000.

[31] Jerome H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[32] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE 63 9*, pages 1278–1308, 1975.

[33] Jonathan S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999.

[34] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.

[35] Jonathan S. Shapiro, John Vanderburgh, Eric Northup, and David Chizmadia. Design of the EROS trusted window system. In *USENIX Security*, 2004.

[36] G. Sirer, M. Fiuczynski, P. Pardyak, and B. N. Bershad. Safe dynamic linking in an extensible operating system. Technical Report TR-95-11-01, University of Washington, 1995.

[37] Michael Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *SOSP-19*, 2003.

[38] Michael Swift, Muthukaruppan, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *OSDI-6*, 2004.

[39] A. Whitaker, M. Shaw, and S. Gribble. Scale and performance in the denali isolation kernel. In *OSDI '02*, 2002.

[40] Maurice V. Wilkes and Roger M. Needham. *The Cambridge CAP Computer and Its Operating System*. North Holland, New York, 1979.

[41] Niklaus Wirth. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, 1992.

[42] Emmett Witchel and Krste Asanović. Hardware works, software doesn't: Enforcing modularity with Mondriaan memory protection. In *HotOS-9*, 2003.

[43] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.