Mondriaan Memory Protection

by

Emmett Jethro Witchel

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2004

Author .....
Emmett Witchel
Department of Electrical Engineering and Computer Science
February, 2004

Certified by .....
Krste Asanović
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Mondriaan Memory Protection

by

Emmett Jethro Witchel

Submitted to the Department of Electrical Engineering and Computer Science
on February, 2004, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Reliability and security are quickly becoming users' biggest concern due to the increasing reliance on computers in all areas of society. Hardware-enforced, fine-grained memory protection can increase the reliability and security of computer systems, but will be adopted only if the protection mechanism does not compromise performance, and if the hardware mechanism can be used easily by existing software.

Mondriaan memory protection (MMP) provides fine-grained memory protection for a linear address space, while supporting an efficient hardware implementation. MMP's use of linear addressing makes it compatible with current software programming models and program binaries, and it is also backwards compatible with current operating systems and instruction sets.

MMP is well suited to improve the robustness of modern software. Modern software development favors modules (or plugins) as a way to structure and provide extensibility for large systems, like operating systems, web servers and web clients. Protection between modules written in unsafe languages is currently provided only by programmer convention, reducing system stability. Device drivers, which are implemented as loadable modules, are now the most frequent source of operating system crashes (e.g., 85% of Windows XP crashes in one study [Swift SOSP '03]). MMP provides a mechanism to enforce module boundaries, increasing system robustness by isolating modules from each other and making all memory sharing explicit.

We implement the MMP hardware in a simulator and modify a version of the Linux 2.4.19 operating system to use it. Linux loads its device drivers as kernel module extensions, and MMP enforces the module boundaries, only allowing the device drivers access to the memory they need to function. The memory isolation provided by MMP increases Linux's resistance to programmer error, and exposed two kernel bugs in common, heavily-tested drivers. Experiments with several benchmarks where MMP was used extensively indicate the space taken by the MMP data structures is less than 11% of the memory used by the kernel, and the kernel's runtime, according to a simple performance model, increases less than 12% (relative to an unmodified kernel).

Thesis Supervisor: Krste Asanović
Title: Associate Professor

Acknowledgments

This dissertation is dedicated to the memory of Joshua Cates (1977-2004). Josh's clear thinking and quick coding contributed greatly to our ASPLOS 2002 paper. He often saw the gaps in my specification of a problem before I did. In addition to possessing great technical skill, Josh also had a great attitude, which made him a pleasure to work with. I was very happy to introduce him to the Jethro Tull album, *Heavy Horses*, which he enjoyed with great relish. The world is a lesser place with his passing.

To the man who helped me find, define, refine and write-up the ideas in my thesis and in several other projects, I would really like to thank my advisor, Professor Krste Asanović. Krste is the man, as is obvious after a few minutes of conversation with him on a startling range of topics. His ability to debug code when told of its failure conditions is scary, and extends to bugs in hardware description languages. His ability to pursue ideas for hours on end has extended beyond my body's ability to maintain sufficient blood sugar levels to my brain.

To my former advisor Professor Frans Kaashoek: who always impressed me with his values system as a researcher, and whose enthusiasm is as impressive as it is infectious. Frans' comments on this thesis were detailed enough to have a permanent effect on my writing style. He also ran the rootinest, tootinest rodeo of a group meeting I've ever been to, which was the location of many memorable graduate student moments.

To Professor Mendel Rosenblum: who took a chance on an unknown, after I pestered him enough; and whose keen intelligence and easy humor were always inspirational. Implementing statistics collection in Embra with him sitting next to me was an amazing experience in externally accelerated coding.

Graduate school is mostly about the other students and the environment, and I am thankful for my time at MIT.

To Ronny Krashinsky: whose originality of thought is startling and whose wealth of ideas makes him a potent ally.

To Chris Batten: who might have attended his lifetime limit of Mondriaan talks, but who always had interesting comments.

To Andy Ayers: a man whose deep insight is only matched by his relaxed nature. He made compiler technology a spiritual art. He taught me about the fat base class problem, among many other gems.

To Chris Metcalf: who opened my eyes to the idea that most problems with UNIX shell scripts and Makefiles come down to levels of quotation.

To Max Poletto: who inspired me with his humility and his use of ASCII pictures as code comments.

To Eddie Kohler: whose scorching mental processes I was able to observe at close range.

To David Mazières: who taught me how to be grateful. I try to be grateful.

More thanks to Chuck Blake, Doug DeCouto, Dan Aguayo, Matt Frank, Mike Taylor, Frank Dabek, Dawson Engler, Emil Sit, Russ Cox, Brian Ford, Michael Kaminsky, Heidi Pan, Ken Barr, Seongmoo Heo, Albert Ma, Jessica Tseng, Michael Zhang, Mark Hampton, Felix Klock, John Jannotti, Debbie Wallach, Fred Chung, John Bicket, Athicha Muthitacharoen,

Sanjit Biswas, Carty Castaldi, and Richard Schooler ("plans don't work, but planning does").

Special thanks to Sam Larson and C. Scott Ananian for their collaboration on papers.

Extra thanks to Benjie Chen for reading a draft of my thesis and providing insightful comments.

Much thanks to Professors Anant Agarwal, Anoop Gupta, Robert Morris, Saman Amarasinghe, and Martin Rinard.

More extra thanks to Professor Barbara Liskov for reading and commenting on my thesis and for sitting on my committee.

I would like to thank the following funding sources: NSF CAREER award CCR-0093354; and DARPA PAC/C award F30602-00-2-0562.

And thank you to Neena Lyall and Shireen Yadollahpour for bringing some administrable order to my unordered travel and reimbursement needs. Thanks to Cornelia Colyer, for administrative help and discussion about France. MIT feels the loss of her death.

I view my dissertation as marking the end of my studenthood. I have long passed the mystery of the 13th grade, and now find myself at the end of the student path. I'd like to thank some of the teachers who got me here.

To Mr. Elfenbien: who taught me science in 5th grade and showed movies of his sky diving adventures, who gave ungraded general knowledge quizzes, one of which contained the question, "What is a female Czar called?" (Czarina).

To Mr. Kennefeck: who taught me English in 7th grade. He would record his lessons before classes every day and then play the tape in each class, occasionally stopping the tape to comment. This started my appreciation of academia's tolerance of eccentricity. Despite his unorthodox methods he was an excellent teacher.

To Mr. Cardinelli: who taught me English in 8th grade and tried to convince me that infer and imply were synonyms. This began my frustration with misunderstanding.

To Dr. Bumby: who taught me math in 10th grade and introduced me to imaginary numbers. I was hooked.

To Mr. New: who taught me physics and AP physics from a midwesterner's perspective. The man liked guns and took some interesting fast-exposure pictures. He gave me a battery tester that I still use today.

To Steve Fisher: who taught my second programming class at Stanford, which was one of the most fun intellectual experiences of my life. He talked fast and had a great feel for the material. I remember him explaining null terminated strings.

To the professor who taught me Modern Algebra (Math 120) at Stanford during the Autumn of 1993: who was human enough to break down in front of the class during the proof of Sylow's theorem and admit that the theorems we were proving were too abstract to relate to anything real, but what he was actually trying to do was teach us how to think.

To my mother: who was involved in my education from the start, often taking a hands on role e.g., by quizzing me on vocabulary words; who related to and encouraged my interest in science; and who endured an extended period of my childhood where my mantra was, "I don't want to go to school," with more patience than should be required of any parent.

To my father: who used to lie on the floor and read the encyclopedia on weekends; who would read a book, listen to the radio and watch TV all at the same time.

To the ancestors: as they were, so I am.

To Dan Yaverbaum: with whom I have asked and answered more questions than anyone else.

To Danielle Adler: with whom I plan to learn the rest of my days.

Special thanks for the inspiration of the art of the Grateful Dead and Bob Dylan.

> For me, pushing myself is way more about, "It's hard to make something that's interesting." It's really, really hard, and I'm sure we don't succeed with every story on every show. Basically, anything that anyone makes... It's like a law of nature, a law of aerodynamics, that anything that's written or anything that's created wants to be mediocre. The natural state of all writing is mediocrity. It's all tending toward mediocrity in the same way that all atoms are sort of dissipating out toward the expanse of the universe. Everything wants to be mediocre, so what it takes to make anything more than mediocre is such a fucking act of will. Anyone who makes something for a living, or even not for a living, if they're really excited about it... You just have to exert so much will into something for it to be good. That feels exactly the same now as it did the first week of the show. That hasn't changed at all. That's the premise of what it takes to make something.

—Ira Glass (producer of National Public Radio's *This American Life*) from an interview in *The Onion*.

> Songs are songs – I don't believe in expecting too much out of any one thing.

—Bob Dylan

> We can only see a short distance ahead, but we can see plenty there that needs to be done.

—Alan Turing

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer architects have been interested in building machines with fine-grained memory protection since the early sixties, in part because of programmer demand. The match between fine-grained memory protection and software engineering seems natural—if the architecture can protect programmer data structures, the architecture (possibly with help from the operating system) can detect and even recover from program errors. Computer architects evolved several designs, none of which fulfilled the promise of fine-grained protection. Segment and capability based machines from the mid-60s to the mid-70s contained ever more elaborate protection mechanisms, but they still failed to provide a satisfactory programming model.

In 1975, Intel began design work on the iAPX 432, an ambitious processor that incorporated some of the most sophisticated architectural support ever conceived for fine-grained protection and object-oriented programming. The processor schedule slipped, and Intel released the 8086 in 1978 as a stopgap. Intel's 8086 (and the 8-bit version, the 8088) became very popular, and when the iAPX 432 finally shipped, its performance was poor. When microprocessors adopted the demand-paged virtual memory hardware pioneered in mainframes in the 60s, they also inherited simple page-based protection schemes. Paging was added to the 80386 in 1985, and page-based protection has been standard for commodity processors ever since. Programmers have accepted the limitations of coarse-grained memory protection, and worked around them, perhaps believing the impressive gains in processor performance since 1985 were possible only with simplified memory protection hardware. The huge volume of code written for a coarse-grained protection model has rendered non-compatible solutions untenable, regardless of their attractive features.

Mondriaan [1] *memory protection* (MMP) provides architectural primitives for fine-grained, hardware-enforced, memory protection that can be efficiently realized in modern processor designs. This support is backwards compatible with existing architectures, operating systems and binaries. Legacy operating systems and software applications written in unsafe languages can use MMP's architectural mechanisms to address the problem of software robustness: finding bugs in development, tolerating them in production, and providing a basis for data security. This thesis contributes a hardware design, and an operating system implementation which validate the long standing insight that fine-grained memory protection is useful to programmers, and which demonstrates the feasibility of efficient hardware

[1] We use the Dutch spelling with the double 'a'

and software implementation.

1.1 The problem of module safety

Modern software development favors modules (or plugins) as a way to structure and provide extensibility for large systems. For example, the Linux kernel has an elaborate module system for device drivers and other kernel subsystems; the Apache web server has an entire web site dedicated to modules (modules.apache.org), which provide such essential functionality as the interpretation of Perl code in web pages [Apa03b]; and the Mozilla web browser supports a general plugin interface to extend the browser with additional functionality like the ability to view documents in Adobe Acrobat format (PDF) [Ado02]. In each case, features are encapsulated into a separate module, reducing the size and complexity of the "core" code. By moving optional features into modules and out of the core, the core becomes simpler. A simpler core is easier to make robust, maintain, and tune for performance. Furthermore, open module interfaces allow third parties to insert their own functionality into a modular system.

One problem with the modular architecture as currently implemented is that there is no isolation between a plugin and the core for modules written in unsafe languages. Most systems (like the ones mentioned) use a single address space and simply link the module into the same address space as the core application. Using a single address space makes communication between the core software and the plugin fast and flexible, but compromises safety. Module boundaries are respected only by programming convention, not enforced by a run-time mechanism. In the operating system, device drivers (implemented as loadable modules) are now the most frequent source of operating system crashes (e.g., 85% of Windows XP crashes in one study [SBL03]).

Without protection, a program error in a module can cause the failure of the entire application. Wild reads, writes, and jumps from a faulty module can cause a system failure, and many different kinds of programming errors in unsafe languages reveal themselves by wild reads, writes, or jumps. The classic example in C, known to even casual programmers, is the NULL memory dereference. Many pointer manipulation errors in C make the program load or store a value to address zero (NULL), which is illegal. The operating system usually halts a program which accesses NULL, and produces a dump of the process' memory for the programmer to analyze. While illegal accesses are frustrating, they provide a fail-stop mechanism. Even more problematic are pointers to legal, but incorrect, addresses which cause data corruption or resource leakage. For instance, buffer overrun attacks overwrite memory locations that should not be writable, and instead of crashing an application, can open a security hole in a web server. In current systems, one module's bad access usually means the entire application must be terminated because there is no way to know what parts of the system have been damaged by the module's failure.

Excessive resource consumption, API violations, and synchronization or locking errors are other possible failure modes, and static analysis techniques can be effective at finding these types of errors [MPC+02, EA03]. Memory corruption across separately loaded modules is one of the most common failure modes, and is not amenable to compile-time analysis. A study of Unix bug databases (including CERT advisories [Sof03]), over the last 10 years shows buffer overrun attacks, which can corrupt stack or heap memory, account for between 25% and 50% of reported vulnerabilities [WFBA00]. Defending against memory corruption and wild jumps requires inspecting every load, store and instruction fetch.

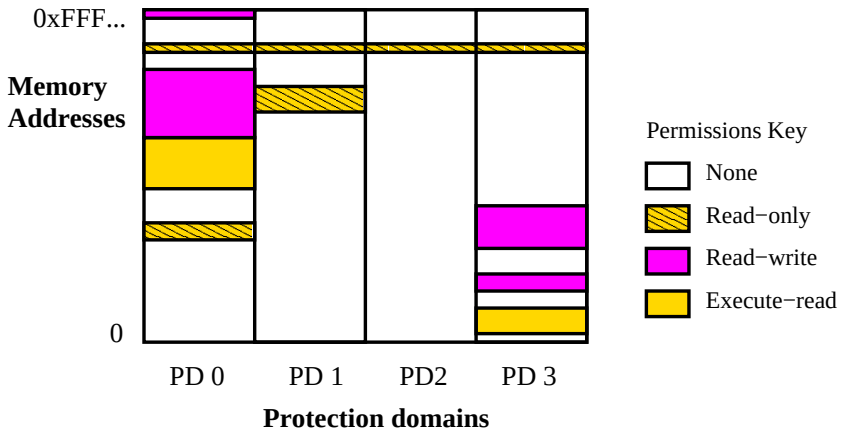


Figure 1-1: A visual depiction of multiple memory protection domains within a single shared address space.

The architects of module-based systems have rejected designs using the native architecture and OS support for a separate address space per module because of the complexity and run-time overhead of managing multiple address contexts. The page-based protection that is ubiquitous in today's hardware is too coarse-grained to support the fine-grained sharing that occurs between modules. Previous hardware mechanisms for fine-grained memory protection, namely segmentation and capabilities, have been plagued by performance problems and an awkward software programming model.

1.2 Fine-grained protection domains

MMP adopts Lampson's term [Lam71], *protection domain*, to refer to a lightweight context which determines permissions for executing code. MMP overlays an address space with multiple, disjoint protection domains, each with a unique set of permissions (see Figure 1-1 which is based on diagrams in [Lam71] and [Lev84]). Each column represents one protection domain, while each row represents a range of memory addresses. The address space can be virtual or physical—protection domains are independent from how virtual memory translation is done (if it is done at all). A protection domain can contain any number of threads, and every thread is associated with exactly one protection domain at any instruction in its execution. Protection domains that want to share data with each other must share at least a portion of their address space. There is no domain-specific portion of an address, a pointer refers to the same memory location in any domain. The color in each box represents the permissions that each protection domain has to access the region of memory in the box. An ideal protection system would allow each protection domain to have a unique view of memory with permissions that can be set on arbitrary-sized memory regions, and would raise a protection exception if the executing thread does not have permissions for an attempted access.

MMP comes close to this ideal. MMP brings increased memory safety to large, legacy systems written in unsafe languages, without decreasing performance significantly. MMP allows a large system, like an operating system, to have several, independent, services which can continue running, even if an individual service fails. Boundaries between services are usually well defined, but often irregular. A service might export ten functions, but also

allow read access to a small, internal, data field for efficiency. MMP supports these irregular interfaces without making programmers change their data layout or code structure.

For example, we modified Linux to run the low-level network driver as a separate service in its own protection domain. If this service tries to violate the permissions it is given, it can typically be killed and restarted; the high-level protocol will resend any packets lost during the failure. A computer system that crashes or becomes unresponsive today could use MMP to become a computer system that tolerated the failure and continued to provide service to the user.

The application examined in detail in this thesis is bringing an increased level of memory safety to the module system found in a modern operating system. Many program errors manifest as memory protection violations, especially if a system is given only the minimal amount of permissions it needs to complete a task. MMP is designed to protect software services from bugs in other services, not malicious code, though it will prevent some malicious attacks. It detects memory use violations, such as one module trying to write another module's unshared data structure, or one module calling another module's private, internal, function. How to recover from a protection violation is a problem that has been addressed by other systems [SBL03], and will not be addressed by this thesis. MMP provides no safeguard against resource exhaustion, thread capture, or denial of service attacks. There are techniques to address all of these problems, but they are beyond the scope of this work.

MMP is not a security system, it is a permissions system. We have designed it to be powerful enough to be the basis of a secure system, but this thesis does not address security policies.

1.3 MMP Overview

MMP consists of hardware and software to provide fine-grained memory protection. MMP modifies the processor pipeline to check permissions on every load, store, and instruction fetch. It supports gate permissions so a service can only be called at an approved entry point (see Chapter 4.3). MMP is designed to be simple enough to allow an efficient implementation for modern processors, but powerful enough to allow a variety of software services to be built on top of it.

The MMP hardware checks the memory accesses and instruction fetches of every thread to see if that thread's domain has appropriate access permissions. Each domain depicted in Figure 1-2, has its own *permissions table*, stored in privileged memory, which specifies the permission that domain has for each address in the address space. The permissions table is similar to the permissions part of a page table, but permissions are kept for individual words in an MMP system. The CPU also contains a hardware control register, which holds the protection domain ID (PD-ID [KCE92]) of the currently running thread, and another register that holds the base address of the active domain's permissions table.

The MMP protection table represents each *user segment*, using one or more *table segments*. A user segment is a contiguous run of memory words with a single permissions value that has some meaning to the user. For instance, a memory block returned from malloc could be a user segment. A table segment is a unit of permissions representation convenient for the permissions table. Different designs for the permissions table break user segments into table segments in different ways.

MMP uses a *protection lookaside buffer* (PLB) to caches permissions information for data accessed by the CPU, avoiding long walks through the memory resident permissions table.

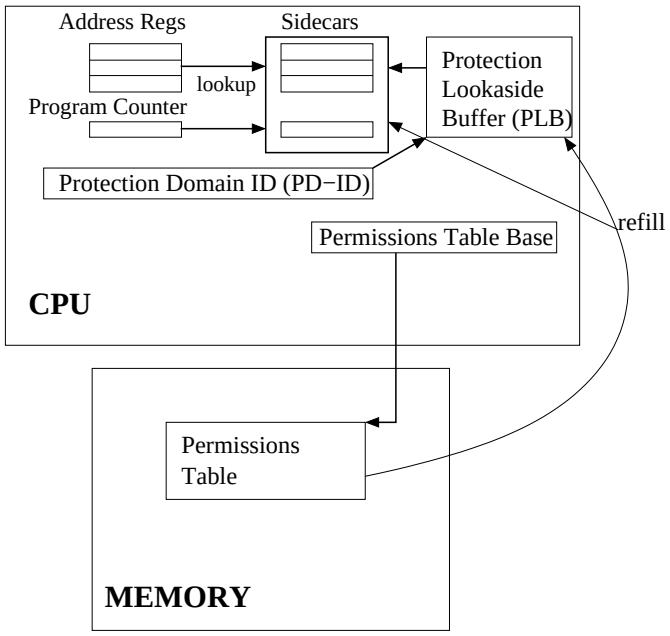


Figure 1-2: The major components of the Mondriaan memory protection system. On a memory reference, the processor checks permissions for the effective address in the address register sidecar. If the reference is out of range of the sidecar information, or the sidecar is not valid, the processor attempts to reload the sidecar from the protection lookaside buffer (PLB). If the PLB does not have the permissions information, either hardware or software looks it up in the permissions table which resides in memory. The reload mechanism caches the matching entry from the permissions table in the PLB and writes it to the address register sidecar.

The PLB resides on-chip with the CPU, and hardware or software reads the permissions table and caches its entries in the PLB. It uses the PLB to cache permissions table entries like a TLB caches translation page table entries. As with a conventional TLB miss, a PLB miss can use hardware or software to search the permission tables.

To further improve performance, and reduce energy consumption, MMP adds a *sidecar register* for every architectural address register in the machine (in machines that have unified address and data registers, a sidecar would be needed for every integer register). The sidecar caches the last table segment accessed through its corresponding address register. The sidecar contains base, bounds and permissions information, obviating the need to access the PLB so long as the addresses generated using the sidecar's address register stay within the table segment. Sidecars save energy because they avoid the associative search performed by the PLB. The sidecars can improve performance because they represent the address range of a table segment exactly, while the PLB index is limited to a power-of-two sized subsegment (see Section 4.2 for details). The information retrieved from the permissions tables on a PLB miss is written to both the register sidecar and the PLB. Sidecars do not complicate the programming model because they are not programmer visible.

MMP incurs both space and time overheads. The space overhead is for the permissions table, which is in main memory and stores permissions information about the virtual address space of a single protection domain. The time overhead is due to the memory references to the permissions table, and instructions for table maintenance. The MMP design reduces

the space overhead by using an efficient multi-level table. The primary cost of the the multi-level table is the number of protection bits per word. The prototype system has 2 protection bits for each 32-bit word. Time overhead is reduced by making the most of the PLB. The permission table uses overlapping, run-length encoded entries. Each entry holds permissions information for adjacent entries, maximizing the utility of loading an entry into the PLB, and reducing the PLB miss rate. Permissions table entries are cached in the data cache, so the main cost of permissions table accesses are cache misses to the table entries.

MMP preserves the user/kernel mode distinction, where kernel mode enables access to privileged control registers and privileged instructions. Access to privileged memory areas (like I/O space) is controlled with MMP. 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. It is special in that it can access all of memory without the mediation of a permissions table.

The software part of MMP is the memory supervisor, discussed in Chapter 6. The supervisor writes the permissions tables, it enforces policy on memory use and sharing, and it provides additional protection services.

1.4 Example and requirements

We provide a brief example to motivate the need for the MMP system, and the specific features it supports. Consider a memory allocation service used by different protection domains. The allocator gets a request for a certain sized memory region, it finds an appropriate block, and then grants permissions on that memory to the calling domain. The domain that allocated the memory can make the memory accessible to another domain. When the memory is freed, the allocator revokes permissions on the region from every domain that has permissions.

Implementing this example requires a memory system to support the following requirements:

- `heterogeneous`: Different protection domains need different permissions on the same memory region to support flexible memory sharing. A domain should be able to give a read-only copy of a dynamically allocated data item to another domain.

- `small`: Sharing granularity can be smaller than a page. Memory allocation systems often allocate a bit more memory than was requested to ease bookkeeping, but page-based allocation would cause too much internal fragmentation to be acceptable for most applications. In an operating system, such fragmentation causes physical memory to go unused.

- `revoke`: A protection domain owns regions of memory and is allowed to specify the permissions that other domains see for that memory, including the ability to revoke permissions. A memory allocation service must revoke access permission when a client frees memory.

- `entry`: A protection domain must be entered at a publicly exported entry point. The called domain should return to the proper location in the caller's domain. A memory allocation service could fail or corrupt its own data structures if a client called a private function that was not intended for public use.

Previous memory sharing models fail one or more of these requirements.

Conventional linear, demand-paged virtual memory systems can meet the `heterogeneous` requirement by placing each thread in a separate address space and then mapping physical memory pages to the same virtual address in each address context. These systems fail the `small` requirement because permissions granularity is at the level of pages. Page-based execute permissions are insufficient to enforce the `entry` requirement.

Page-group systems [KCE92], such as HP's PA-RISC and IBM's PowerPC, define protection domains by which page-groups (collections of memory pages) are accessible. Every domain that has access to a page-group sees the same permissions for all pages in the group, violating the `heterogeneous` requirement. They also violate the `small` requirement because they work at the coarse granularity of a page or multiple pages. Domain-page systems [KCE92] are similar to our design in that they have an explicit domain identifier, and each domain can specify a permissions value for each page. They fail to meet the `small`, and `entry` requirement because permissions are managed at page granularity.

Capability systems [DH66, Lev84] are an extension of segmented architectures where a capability is a special pointer that contains both location and protection information for a segment. Although designed for protected sharing, some of these systems fail the `heterogeneous` requirement for the common case of shared data structures that contain pointers. Threads sharing the data structure share its pointers (capabilities) and therefore see the same permissions for objects accessed via those capabilities. Some systems (e.g., EROS [Sha99]) remove this restriction by supporting a special capability modifier which downgrades permissions when capabilities are read [Sha99].

Many capability systems fail to meet the `revoke` requirement, because revocation can require an exhaustive sweep of the memory in a protection domain [CKD94]. Some capability systems meet the `heterogeneous` and `revoke` requirements by performing an indirect lookup on each capability use [HSH81, SSF99], which adds considerable run-time overhead. Special capability types can enforce the `entry` requirement exactly.

1.5 Contributions of the thesis

The contribution of this thesis is to provide a hardware/software design for fine-grained memory protection that has several attractive features:

- **MMP memory protection primitives naturally fit software usage patterns.** The author's ability to adapt quickly several major Linux subsystems to use MMP provides evidence that MMP's abstractions fit those already present in software.

- **MMP is backwards compatible with current software.** We did not change most of Linux to enable it to use MMP, because MMP does not force us to change code unrelated to protection. For most system utilities, the MMP-enabled Linux uses the same binaries as Linux.

- **MMP is compatible with current instruction sets.** Our Linux prototype runs on the x86, which is not a cleanly designed instruction set. However, MMP does not need instruction set support, and it can cope with irregularities like byte-aligned instructions.

- **MMP is easily implemented in high-performance hardware.** Memory protection checks are not on an instruction's critical path, allowing the check to overlap with instruction execution.

MMP succeeds where previous designs failed because it separates protection metadata from program data, allowing permissions to be compressed and more effectively cached. This is similar to paging hardware, where address translation separates the translation metadata from program data and compresses it, representing an entire page's offset with a single value. MMP does not get in the way, maintaining backward compatibility with instruction sets, operating systems, and programming models. It also does not contain any confusing new programmer-visible abstractions, just permissions on words of memory.

In this thesis, the main application for fine-grained protection is bringing memory safety to operating system modules written in unsafe languages, though Chapter 10 discusses many other uses.

We validate our design of the permissions table and caching structures (PLB and sidecars) by evaluating their performance on user-level applications written in C and Java. The results from Chapter 5, show that MMP has little performance penalty when it is used to protect large memory regions. We then use it for fine-grained protection by placing inaccessible (guard) words before and after every dynamically allocated block of memory (every call to malloc). We measure a space overhead of under 9%, while adding fewer than 8% additional memory references (application references and permissions table references divided by application references).

We demonstrated MMP's backwards compatibility and ease of use by adapting Linux 2.4.19, executing on Intel's x86 architecture, to use it. Linux is a mature operating system, and the x86 is a well-entrenched architecture. The parts of the kernel that deal with memory permissions were the only ones modified, and these were modified to increase the memory isolation between kernel subsystems. We modeled the MMP hardware, and built the memory supervisor, which is the software that manages the hardware and provides a useful interface to the rest of the kernel. We modified Linux's memory allocators, and its system utilities to isolate kernel modules in their own domain, and make kernel-module (and inter-module) memory sharing explicit. We call this modified system Mondrix. That a single programmer could adapt Linux to use MMP provides some evidence for our claim that MMP provides a useful software abstraction. We then simulated the MMP hardware within bochs [Sou03], which is a full x86 machine simulator, sufficient in detail to boot and run an operating system.

Mondrix requires less than 11% additional memory space, and adds less than 12% execution cycles (according to a simple performance model) to a range of OS intensive workloads. The evaluation details are in Chapter 7, and they show that OS support for MMP is efficient, and that making memory sharing explicit with MMP, even in the context of a large and complicated code base, carries a modest performance cost.

Fine-grained memory protection has been desired by researchers in commodity hardware for years [AL91]. It is useful for stopping buffer overrun security attacks [WFBA00], data watchpoints [Wah92], and generational garbage collectors [LH83]. These and other applications are discussed in Section 11.1, but the most important uses might not yet have been conceived. In the author's experience, when told about MMP, most researchers confess some interesting design idea they abandoned for lack of hardware and OS support for fine-grained protection.

1.6 Thesis outline

This thesis is structured as follows. Chapter 2 gives an overview of related work, examining previous forms of architectural support for fine-grained memory protection, and software solutions to memory safety and system extensibility.

Chapter 3 explains the MMP permissions table, the main data structure written by software and read by hardware that provides the permissions information which is checked on every load, store, and instruction execution. Its design is similar to a page table, but with only protection information. Chapter 4 discusses the hardware structures, which cache the permissions table entries. These two structures make a complete system, which Chapter 5 evaluates for different kinds of user-level programs written in C and Java.

Chapter 6 describes the MMP memory supervisor, which is the kernel code that manages the permissions tables and provides additional memory protection services. Chapter 7 describes the Linux prototype, called Mondrix. We modified the bochs [Sou03] x86 emulator to model the MMP hardware, and adapted the Debian distribution of Linux 2.4.19 to use the modified hardware. We wrote a memory supervisor for Linux, and added inter-module protection for the EIDE disc driver, the network driver, and other necessary kernel modules. We present a detailed performance analysis of the prototype in Chapter 8.

Chapter 9 presents a design for extending MMP memory isolation to stack memory. Chapter 10 discusses how the MMP framework can be extended to memory metadata beyond permissions information. It explains and evaluates fine-grained translation which is useful for zero-copy networking. Chapter 11 describes many additional applications for fine-grained protection and translation. It goes on to discuss programming language support for MMP, and concludes.

Chapter 2

Memory Protection

This chapter reviews how other systems provide memory protection. The page-based protection that is nearly ubiquitous today has shortcomings because of its coarse granularity. Fine-grained protection has long been identified as useful [Bur61, DH66, Sal74], but previous hardware support for it (segmentation and capabilities) have been plagued by performance problems and an awkward software programming model. Software-only approaches generally restrict implementation to a single language; they cannot make use of legacy code (especially code written in unsafe languages) and third party modules distributed in binary form. They also often suffer from excessive CPU and memory usage.

2.1 Page-based protection

Memory protection was originally introduced to ensure safe multiprogramming of time-shared computers in the late 1950s. Page-based virtual memory systems, introduced with the Atlas [Fot61], have become the dominant form of memory management in modern general-purpose computer systems. Modern architectures and operating systems have moved towards linear addressing, in which each user process has a separate, linear, demand-paged, virtual address space. Each address space has a single protection domain, shared by all threads that run within the address space. A thread can only have a different protection domain if it runs in a different address space. Sharing is only possible at page granularity; a single physical memory page can be mapped into two or more virtual address spaces. A user/kernel processor state bit provides protection between any user task and the kernel.

Although linear, page-based addressing is now ubiquitous in modern OS designs and hardware implementations, it has significant disadvantages when used for protected sharing. Pointer-based data structures can be shared only if all words on a page have the same permissions, and the shared memory region resides at the same virtual address for all participating processes. The interpretation of a pointer depends on addressing context, (i.e., addresses used by one context can be invalid in another) and any transfer of control between protected modules requires a context switch, which is expensive in modern processors. The coarse granularity of protection regions, the high cost of protecting memory via system calls, and the overhead of inter-process communication limit the ways in which protected sharing can be used by application developers. Although designers have been creative in working

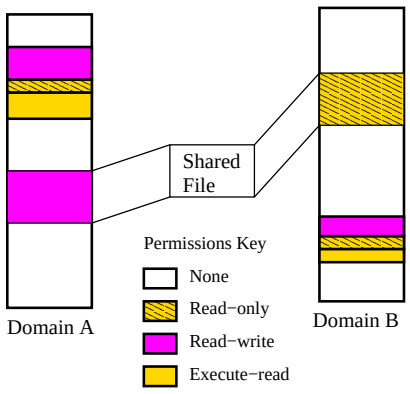


Figure 2-1: Sharing memory at page granularity via `mmap`. A file is mapped into domain A with read-write permissions, and into domain B with read-only permissions. In this example, each domain is a process.

around these limitations to implement protected sharing for some applications [Chu96, Lie95, HHL+97], each application requires considerable custom engineering effort to attain high performance. In practice, designers of web browsers or kernel modules have sacrificed robustness in favor of performance by foregoing hardware protection and placing all modules in the same address space [Apa03a, Moz03, Tor03].

Figure 2-1 shows an example of page-based memory sharing using `mmap`. A single file is mapped into two different domains, at different addresses. It is a strength of `mmap` that Domain A can have read-write permissions and domain B has read-only permissions. The shared region must be an integral number of pages, which rules out using page based protection to share most data structures found in an operating system without using additional padding (which wastes physical memory). In this example, the two domains have mapped their shared regions at different addresses, which means they cannot share pointers. While `mmap` has a `MAP_FIXED` option to allow a memory mapping at a fixed location, a programmer might find it difficult to find a suitable shared location in two unrelated processes.

2.1.1 Page sharing

Some architectures support page sharing. Address space identifiers (ASIDs) in the MIPS [KH92] and UltraSPARC [Sun96] architecture are process tags for TLB entries. This design allows two different processes to have a mapping for the same virtual address in the TLB at the same time—the ASID distinguishes them.

Every process in the UltraSPARC (and PA-RISC [Hew02]) has a set of ASIDs, which provides support for groups of processes sharing a small number of pages or page groups. In Figure 2-1, if the shared file were mapped at the same location and with the same permission in domains A and B, the TLB entries for the shared file could have ASID C, which is an extra ASID shared by both A and B. While memory sharing is important enough to warrant architectural support in modern architectures, the page granularity limits its usefulness to software.

2.1.2 Grouping pages

Several systems provide support for aggregating page-based protections. *Domain-page* systems [KCE92] can set permissions only at the granularity of a memory page. Page group systems, such as HP PA-RISC and PowerPC, require that a collection of pages are mapped together and with the same permissions across all domains (though each domain independently might or might not have access at that fixed permission) [KCE92]. The Apple Newton [SW92, WSW+94] has a form of page group system, where an active process has access to a set of regions (called domains) which have the same access permissions across all processes.

MMP can be considered a *domain-segment* system, because it allows permissions to be set on arbitrary runs of words.

2.2 Segmentation

Segment-based schemes were among the first approaches to provide fine-grain cooperation and information sharing between processes. The Burroughs B5000 [Bur61] was one of the first machines to offer a segment-based protection scheme. An address is not simply an index into a physical array of bytes, but rather the top bits of the address indexes a segment table, which provides a base address of a segment, and the bottom bits of the address provide the offset in that segment. Entries in the segment table include a field describing the protection for the segment. A program can have many variable-sized code and data segments, and a single stack segment. A hardware register points to a program reference table (PRT), which holds an array of segment descriptors for the program. Every user code or data reference must indirect through this table, and the processor checks accesses against the base and bounds and access permissions held in the segment descriptor. Only operating system code updates the PRT. Segments can be shared between processes by mapping the same descriptors into two different PRTs, but the shared descriptor must reside at the same offset in both PRTs. Fabry [Fab74] discusses the difficulties of allowing more flexible sharing of segments between processes.

One disadvantage of segment-based schemes is the need to divide addresses between segment numbers and segment offsets, requiring a trade-off between the number of segments, and the maximum size of a segment. This static partitioning also complicates scaling the architecture to a larger address space, as seen in the evolution of Intel's x86 [Int02].

The systems built on segment hardware, preeminently Multics [Sal74], are similar to the systems we hope to encourage with MMP. They feature hard modularity where "programming errors related to using incorrect addresses tend to be immediately detected as protection violations, and do not persist into delivered systems." [Sal74] While the goal is the same, MMP is different from segmentation; MMP maintains linear addressing and is compatible with high performance, modern architectures.

One big drawback of using segmented addressing is that it exposes hardware detail to programmers and compilers in an inconvenient way. The classic example is control transfer on Intel's x86. With 16-bit code segments, the programmer or compiler must know if a procedure call could use a near pointer (within the segment) or a far pointer (outside the segment). Foisting this addressing complexity on users is an unpopular feature of the architecture. In MMP, the management of the protection structures is not seen by the user.

Modern architectures like PowerPC [IBM02] and HP's PA-RISC 2.0 [Hew02] have adopted

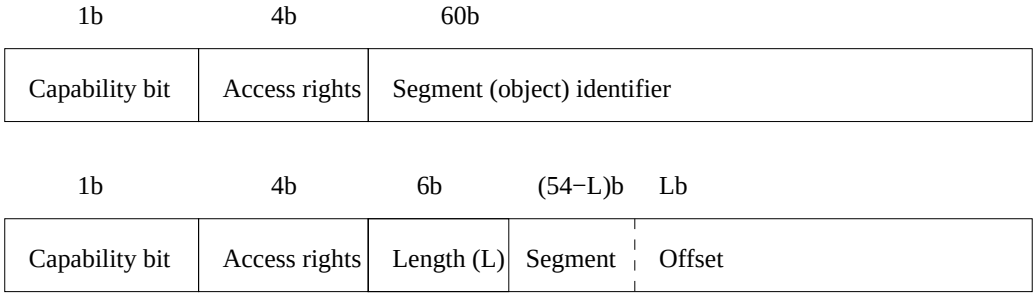


Figure 2-2: Two example capabilities. Both example capabilities contain a capability bit, which distinguishes capabilities from non-capabilities, and is set or cleared by privileged instructions. Both capabilities contains four bits of access rights (e.g., read/write/execute/enter), which indicate the operations on the object allowed by the capability. The top capability, common in older capability designs, contains a segment or object identifier which must be interpreted by the hardware, and combined with an offset, to generate a virtual address. The bottom capability (used in the M-machine [CKD94]) contains a virtual address, and features a variable division between segment identifier and segment offset. The capabilities are 65 bits because the capability bit is not architecturally visible.

segmentation as a way of expanding the virtual address space (from 32 to 64 or 96 bits in PA-RISC). The expanded addresses facilitate sharing among cooperative processes and obviates the need to flush the TLB on every context switch. These processors use the upper bits of the address to choose the segment, which means that segments sizes are large (256MB for PowerPC). Our segments are arbitrarily sized, and much of our interest is in the smaller sizes.

2.3 Capabilities

Capability-based architectures [DH66, Lev84] are an evolution of segment-based schemes [SS75], where a capability is a special pointer that contains both location and protection information for a segment (for examples, see Figure 2-2). The data that a user program can access is defined by the set of capabilities it possesses, and memory references are always made relative to a given capability. Hardware-based capability systems require instruction set support.

To protect their integrity, capabilities are modifiable only by the OS kernel. This property is ensured either by using special bits to tag each memory word holding a capability [WN79, CKD94] or by placing capabilities in protected memory segments (usually called C-lists) [DH66]. Tagging wastes memory bits on non-pointers.

Figure 2-2 shows two styles of capability representation (both are tagged). The first is common in early capability machines like the CAL-TSS system [Lev84], the Cambridge CAP [Lev84], and Multics [Sal74]. Capabilities for these systems do not contain addresses, but rather contain an identifier which is an offset into a global or local segment (or object) table. The table contains the virtual (or physical) memory address, and length information for a memory segment. Some machines (e.g., the Cambridge CAP) have user-invisible registers which cache the result of resolving a capability to the memory address of the

segment it protects, so the resolution does not need to be performed on every memory access (MMP's sidecar registers are analogous to these registers). This style of capability can contain base and length information, to enable a capability to grant access rights on a subsection of the segment to which it refers. A register or instruction immediate provides the offset within the segment referred to by the capability.

The bottom capability, used in the M-machine [CKD94] removes the extra levels of indirection going from capability to address that are required by the top style of capability. The M-machine capability contains a segment number and offset, which specifies a byte location in memory. It allows a floating division between segment number and offset, so memory can be divided into any number of power-of-two sized segments. Capabilities are created to protect a segment of memory, so arithmetic operations on them check that the resultant address still points within the original segment.

Most systems allow memory data to become persistent, so capability systems need some way to make capabilities persistent. Capability-based access control is different from the access control list (ACL) model used by traditional file systems, so many capability-based systems [WCC+74, JJD+79, Ber80, SSF99] do away with traditional file systems. The temptation to control everything: memory regions, disc storage, even I/O devices with capabilities demonstrates their power, but it also makes resource management for capability-based systems different from non-capability-based systems. Code written for a non-capability system often must be redesigned and reimplemented for use in a capability system, raising the barrier for entry.

Capabilities can be freely exchanged between processes, but once granted to a process they are difficult to revoke. Revocation can require either an exhaustive sweep of the memory in a protection domain [CKD94] or indirection on every pointer access [HSH81, Lev84]. IBM's AS400 provides revocable and non-revocable capabilities because of the runtime cost of supporting the revocation of a specific capability. Even if a capability can be revoked, if it is stored in a data segment shared by many processes, it is difficult to revoke access rights for one process without affecting other processes sharing the same data.

Other disadvantages of capabilities are that they require more space than conventional pointers because they often hold base and length information in addition to permissions. This disadvantage can be mitigated if segment sizes and alignment are restricted, for example, to power-of-two sizes [CKD94]. But the bits used for protection and other metadata must come from somewhere, so they either increase the size of a pointer, or they reduces the size of the address space (from 64 to 54 bits for the design in [CKD94]).

Capability systems have problems with allowing different domains to have different permissions on a capability-based data structure. Since the capability itself holds the access permissions, it requires additional capability permission types for a domain to export a read-only version of a data structure containing capabilities (i.e., pointers), when the domain itself wants to retain read-write permissions. This problem is solved in EROS [SSF99, Sha99], by additional capability types. EROS is a software-based capability system, allowing it to add capability types more easily than hardware implementations. It is discussed below under software techniques.

MMP provides many of the same benefits of capabilities, primarily the ability to give a task only the minimal set of memory permissions it needs, while avoiding most of their disadvantages.

2.4 Embedded systems

Another important application area for single address space techniques is embedded systems, which often have only a single small physical address space. These systems commonly consist of several closely interacting threads. Protected sharing is important enough to merit architectural support in these systems. The ARM940T is a recent embedded processor that allows the active process to access 8 overlapping segments of the global physical address space, but the segments must have power-of-2 size and alignment with a minimum size of 4 KB [ARM00]. The ARM 1156 decreases the minimum size to 32 bytes [Lev03], indicating the need for embedded processors to support fine-grained memory protection.

2.5 Software techniques

There are a range of software techniques for memory protection. Most of them provide higher-level safety guarantees than MMP, and all of them could use MMP facilities to improve functionality or performance.

2.5.1 Nooks

Nooks [SBL03] provides device driver safety using conventional hardware. It shares the MMP design goal of guarding against programmer error rather than malicious code, but it achieves this goal without adding hardware mechanism, unlike the MMP system. Nooks uses conventional paging hardware to isolate modules by putting them in different addressing contexts (protection domains). All of these domains execute with full kernel privileges, but they differ in their view of memory permissions.

Nooks is limited by the coarse-granularity of page-based protection, and the run-time costs of switching addressing contexts. These factors sometimes force Nooks to place several modules in the same protection domain. For example, many drivers are split into two modules, a top half which manages device-independent algorithms, and a bottom half with device dependent functionality. Nooks places both halves in the same domain, because it must limit the number of Nook boundaries crossed during execution to maintain good performance. MMP can enforce the natural module boundaries established by the Linux kernel developers. MMP enables much finer-grained modules divisions, and defers to the programmer, placing each module in its own domain. As a consequence, the frequency of cross-domain calls in the MMP system (Section 8.5) is at least an order of magnitude greater than Nooks [SBL03] without a decrease in performance, indicating that MMP offers greater modularity and protection.

Nooks is an elegant solution to the specific problem of bringing safety to OS extensions. MMP is a general purpose architectural mechanism applicable to the problem of safe OS extensions, and also to safe user extensions, and a variety of other applications like data watchpoints, optimistic compiler optimizations, and efficient write barriers for garbage collection (see Section 11.1).

Nooks includes a recovery system, which tracks kernel objects and tries to reclaim resources on a fault. Mondrix does not have a recovery mechanism. If recovery can be done at a coarser level of granularity than isolation, Mondrix can use many of Nook's techniques.

2.5.2 Safe languages

SPIN [BSP+95] is perhaps the largest OS project to have examined a safe language as the primary extensibility mechanism. SPIN shows 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 [BSP+95]. One advantage of MMP is that it supports legacy code, written in unsafe languages.

Another problem with language-only safety is the size of the system that must be trusted. A complete language compiler and runtime, especially an optimized system which employs complex analyzes to improve runtime efficiency, is a large and complicated code base, all of which must be trusted. For an MMP system, one need only trust the MMP hardware and the MMP supervisor software. These components are likely to be simpler and more amenable to verification than a complete language compiler and runtime.

There are other safe language approaches (e.g., [JMG+02], [vECC+99]) for OS extensibility and they generally have the same problems—excessive CPU and memory consumption is common in safe languages or unsafe languages retrofitted with type information. A safe language restricts an implementation to a single language; it ignores a large base of existing code; the analysis needed to establish type-safety can be global and thus difficult to scale; and type-safe languages often need unsafe extensions to manage devices.

2.5.3 Software capability systems

EROS [SSF99, Sha99] is a capability system built on the Intel x86 architecture, whose emphasis is on software security. Capabilities allow for careful analysis of access rights, and Shapiro provides a formal model that allows programmers to prove that a given capability system implements a specified security policy [Sha99].

EROS uses the memory protection mechanisms of the x86. Its capabilities protect coarse-grained objects, (e.g., processes), short capability lists (called nodes), or memory pages. Unlike capability systems with a primarily hardware implementation, EROS does not have fine-grained memory protection as a design goal. If EROS were implemented on an MMP-enabled processor, it could support finer-grained memory objects.

EROS addresses some longstanding problems with capability systems. It supports a variety of capability types, because its software implementation allows this kind of flexibility. EROS has a take capability, which confers the right to read a capability from another domain, and a diminished take capability, which confers the right to read a capability from another domain, but read-write capabilities are downgraded to read-only capabilities as they are read. This allows one domain to export a read-only copy of a data structure to another domain. EROS also supports the opaque modifier on capabilities, which allows domains to use metadata, like mapping tables, without being able to read or write them.

Capabilities are efficiently implemented on the x86 by having an optimized representation for capabilities that are in memory. Global revocation for a capability is efficient because the capability has a version number which can be incremented to invalidate all on-disc copies. Selective revocation is accomplished by translucent forwarding [Red74] where forwarding occurs only across compartment (protection domain) boundaries. Forwarding cycles are prevented by setting a fixed limit to the length of a forwarding path.

2.5.4 Single address space operating systems

Single-address space operating systems (SAS OSes) place all processes in a single large address space [Cha95, HEV+98], and many use protection domains to specify memory permissions for different thread contexts [KCE92]. The granularity of protection in single-address space systems is usually a page to match the underlying paging hardware. MMP builds upon the SAS OS protection domain approach but extends it to word granularity.

2.5.5 Static analysis and model checking

Modern static analysis [WRBS00] and model checking tools [ECC01, MPC+02] can scale sufficiently to deal with large OS codes. These systems can find many important bugs without flooding the user with false positives. Model checking is primarily useful for checking consistent use of APIs, such as locking primitives, and does not try to guarantee memory safety at runtime.

Static analysis can reveal some memory use problems like reading uninitialized memory, but classic static analysis suffers from finding problems that never actually happen (since it's not possible to enumerate the feasible paths). They are also notably limited in their ability to deal with concurrent behavior.

The ability of static analysis to verify dynamically changing roles for memory is limited. For instance, after the network driver copies an incoming packet to a kernel-supplied buffer, the Mondrix networking code in the kernel takes away write permissions from the network driver on the packet memory. If the driver code saved a pointer to the old packet buffer, it would be very difficult to prove statically that the driver code does not accidentally dereference the saved pointer when it is called to receive the next packet. With MMP, the kernel revokes the driver's ability to write the old packet, so any attempt to do so will cause a memory fault. Proving that kind of safety property statically is beyond the means of current techniques.

Finally, modern static analysis packages try to focus on feasible execution paths, sacrificing soundness for scalability. These systems suffer from false negatives. False negatives from static analysis could be caught by the dynamic checking done by the MMP system.

2.5.6 Lightweight remote procedure call

Lightweight remote procedure call (LRPC) [BALL89] enables modular boundaries for unsafe languages, using a software-enforced discipline for protected calling. It allows the partitioning of an OS into different protection domains whose interactions are protected, but LRPC achieves this protection by using data marshaling and copying, a costly process which MMP avoids. Data copying is inefficient, and imposes a minimum size on a protection domain so calls to the domain can be amortized. MMP cross-domain calls (Chapter 4.3) are analogous to light-weight remote procedure calls, though cross-domain calls do not require copying data for protection, or an argument stack per domain pair, as LRPC does.

2.5.7 Software fault isolation

Software fault isolation [WLAG93] is a general technique that restricts the address range of loads and stores by modifying a program binary. Purify [Rat02] is a commercial software product for memory bounds checking based on executable rewriting. It has gained wide

acceptance, however it can't be used in an OS kernel, or in some embedded development environments. These environments lack required system services (like files), and the allocators for these systems tend to have individual, non-standard semantics. Purify can degrade performance considerably (9 to 29 times according to one report published by Rational Software, the company that makes Purify [Nat97]).

2.5.8 Proof-carrying code

Proof-carrying code [Nec97] is a system where software carries its own proof of safety. The proof is checked at run-time. A proof checker is simpler than the system which produces the proof, and a client of the system need only trust the checker, avoiding the complexity and performance problems of safe language systems and software fault isolation. It can prove that a native code program, compiled from Java, is correctly typed according to the Java type system, and correctly uses dynamic dispatch and exception handling. This approach scales to Java programs up to a half a million lines [SN02]. The proofs in proof-carrying code are verified statically, so they are limited to the same class of safety properties that static analysis can verify.

2.6 Protecting control flow

An important function of a protection system is to enable protected subsystems [Sal74], which are collections of code and data that can be called only at exported entry points. In Linux, modules are protected subsystems, and so are the parts of the kernel that deal with formatted output, like `printk`, `sprintf`, etc. Subsystems expect other subsystems to call only certain functions, though the set of exported functions is not uniform across all other subsystems. For instance, a device-independent Linux network driver might expect the kernel to call a few of its functions, but it might expect the device-dependent portion of the driver to call many more of its functions.

Mechanisms for protected control transfer allow one subsystem to call another only at approved entry points, and they make some provision for the callee to have permissions on function arguments. Heap-based function arguments are usually protected by the standard mechanism of the system: segment descriptors for segmented architectures; capabilities for capability-based architectures; marshaling for inter-process communication based systems (e.g., microkernels); and protection tables for MMP. Several systems allow function arguments allocated from stack memory, and a protected call mechanism must make special provision for these function arguments.

2.6.1 Gates

Several architectures [Int97, Hew02], use *gates* to change protection domains. Gates are a hardware mechanism to transfer control between different protection domains. For example, they are used by HP-UX to implement system calls. Intel's x86 and Itanium architecture use gates as a general call mechanism, but one primarily intended to allow inter-privilege-level calls. The segment selector for the call specifies the privilege level and the call gate, which holds the new code segment and program counter value. The x86 call gate also switches

stacks if the call changes privilege level, and it copies arguments from the old stack to the new stack.

The x86 call gate is unpopular with operating system developers because its performance is equivalent to a software implementation, but it lacks software's flexibility. Most operating systems use a simple interrupt instruction to implement a system call, and most use only two of Intel's four protection levels.

Multics [Sal74] is built on hardware that supports call gates, and the gates are used to build protected subsystems in a style similar to what MMP enables. Multics gates are enforced using segmentation; part of a segment's access bits specify that code from one segment can call code from a different protected subsystem. Multics limits the number of subsystems in a process to 8, and only allows a subsystem to call another with a higher identifier.

2.6.2 Protecting control flow with capabilities

The Cambridge CAP computer has enter capabilities, which allows the holder to call a service at a given entry point. It also has a hardware managed cross-domain call stack, and instruction set support for protected control transfer [Lev84]. EROS [Sha99] also has enter capabilities. When code executes an enter capability, a resume capability is synthesized by the system, which allows the processor to return to the correct location in the caller's subsystem. Resume capabilities can be copied, but whenever a thread uses any of them, they are all revoked, insuring that each call has only a single return.

In capability-based systems, all pointers are capabilities, so all parameters are capabilities. No special handling for stack allocated storage is required.

2.6.3 Microkernels

Microkernels use different address spaces for different subsystems, ensuring isolation, but increasing the cost to move from one subsystem to another. In these systems, inter-process communication (IPC), moves a thread from one subsystem to another. IPC is usually implemented as a remote procedure call [Nel81] (RPC) optimized for a single machine. The RPC mechanism insures that control is transfered only to a subsystem's exported entry point. All arguments, whether allocated from stack or heap memory, are marshaled. Most of the time, marshaling arguments means the caller copies them into a message buffer, and the callee copies the message buffer to its own data structures. Copying is not necessary if the RPC system places the arguments in registers, or uses page remapping to make the argument memory available directly to the callee address space.

L4 and related systems showed that an IPC with register arguments can be as fast as 180 cycles on an Intel Pentium III, 450MHz [HHL+97, HLP+00]. While this performance penalty is low, it is not low enough to put each kernel module in its own protection domain. Nooks (see Section 2.5.1) uses an even more highly optimized IPC mechanism, which can be more efficient because unlike L4's IPC mechanism, it is not safe (the kernel protection domains can subvert the protection mechanism). While Nook's IPC mechanism is efficient, as we noted above, Nooks often places several drivers in the same Nook to reduce the cost of cross-domain transfers.

2.7 Summary

This chapter positions MMP in the taxonomy of hardware support for fine-grained memory permissions. MMP offers finer granularity and a more flexible sharing model than page-base protection. MMP is similar to segmentation and capabilities in that it allows the user to manipulate permissions on memory regions of very different sizes, but it maintains linear addressing and compatibility with current ISAs, programing models, and even program binaries. As a simple hardware mechanism, it avoids the complexity and performance problems with all-software approaches to memory safety.

Chapter 3

MMP Permissions Table

This chapter discusses the format of the permissions tables. A permissions table specifies a permissions value for every word of an address space. The chapter starts with descriptions of different table organizations, and concludes with possible refinements on our design.

The main operation the permissions table must support is finding the access permissions for a given address. The goal in the design of the permissions table is to balance the efficiency of the lookup with the extra storage needed for the table. The permissions table specifies the actions (execute, read, write, none) that may be performed on each 32-bit word in memory (see Figure 1-1). MMP checks that every word loaded, stored or executed by the processor has load, write, or execute permission (respectively) in the permissions table. In addition, MMP checks that entry to and exit from protected procedures occurs at properly marked instructions.

Permission Value	**Meaning**
00	no perm
01	read-only
10	read-write
11	execute-read

Table 3.1: Example permission values and their meaning.

All the designs discussed in this chapter provide two bits of protection information per 32-bit word, whose interpretation is shown in Table 3.1. Although we focus on 32 bit words and 2 permission bits, MMP works for any chosen set of permission values, and any address size. However, space and access time are likely to increase if more protection bits, or longer addresses are used. We discuss alternatives for more protection bits and longer addresses in Section 3.4.3.

While two bits of permissions can protect data, and can distinguish code form data, MMP provides *gate* permissions for safe cross-domain calls. Gates [Sal74, Int96, Hew02, Int02] are a mechanism which allows control to flow from one privilege level to another. An MMP gate marks an instruction as a valid entry or exit for cross-domain control flow. Proper use of gates by the programmer allows the hardware to check that cross-domain control flow only happens to the entry point of exported functions, and the hardware can check that if the function returns, it returns to the location and domain where it was called.

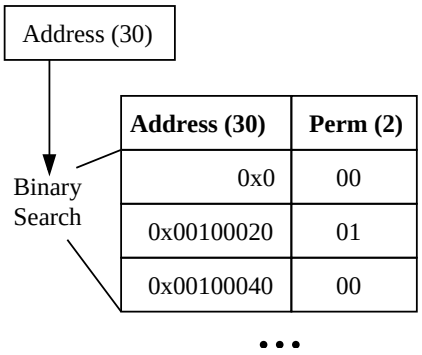


Figure 3-1: A sorted segment table (SST). Entries are kept in sorted order and binary searched on lookup. In this example, there is a single read-only region from `0x00100020` – `0x0010003F`.

Reducing the number of protection bits per word stored in the protection table is the main technique for reducing its size. Consequently, gate permissions are not represented as another permissions value, since that would increase the number of protection bits per word to three. Other properties of gates, primarily their sparsity, and the additional data they require, further justify segregating them from the main protection table. Section 3.3 discusses the gate tables.

The permissions table resides in protected system memory (like a page table), and it is cached by the processor's data cache (if the processor has a data cache).

The first encoding we study is the *sorted segment table* (SST), which is easy to understand, and it is used by the MMP software described in Chapter 6. When it has many entries, looking up a particular entry is slow, so we next examine a *trie*, which does not have this problem since a lookup accesses at most a fixed number of table entries. We present a trie with two different entry formats—bitvectors and run-length encoded entries. Finally, the chapter concludes with possible enhancements to the table encoding.

3.1 Sorted segment table

A simple design for the permissions table is a linear array of segments ordered by segment start address. A segment is any number of contiguous words (starting on a word boundary) with the same permissions value. MMP segments are simply a data structure for the MMP permissions table, they are not part of the user-visible architecture, as they are in segmented address architectures (described in Section 2.2). Figure 3-1 shows the layout of the sorted segment table (SST). Each entry is four bytes wide, and includes a 30-bit start address (permissions granularity is a 4-byte word, so only 30 bits are needed) and a 2-bit permissions field. The start address of the next segment implicitly encodes the end of the current segment, so segments with no permissions are used to encode gaps and to terminate the list. To find the permissions of an address, MMP uses binary search to locate the segment containing the demand address.

The SST is a compact way of describing user segments, especially when the number of user segments is small. A user segment requires only 32 bits to represent, if it abuts another segment. If it does not abut, two entries can represent it, e.g., `<0x00100020, 01>` and `<0x00100040,00>` in Figure 3-1. The problem with the SST is that it takes $O(log(N))$

memory references to find the entry for a given address, where N is the number of user segments in the table. Because the entries are contiguous, they must be copied when a new entry is inserted, requiring $O(N)$ memory references on every table update. Finally, the SST table can only be shared between domains in its entirety, i.e., domains either share no entries from their SST tables, or they share their entire tables.

3.2 Trie

A trie stores data like a traditional forward-mapped page table (which itself is an instance of the more general trie data structure). The top bits of an address index into a table, whose entry can be a pointer to another table which is indexed by the most significant bits remaining in the address.

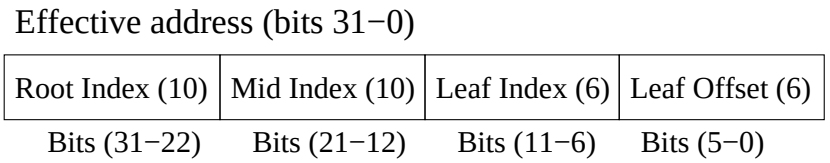


Figure 3-2: How an address indexes the trie.

The trie table is organized like a conventional forward mapped page table, but with an additional level. Three loads are sufficient to find the permissions for any address. Figure 3-2 shows which bits of the address are used to index the table, and Figure 3-3 shows the lookup algorithm. Entries are 32-bits wide. The root table has 1024 entries, each of which maps a 4 MB block. Entries in the mid-level table map 4 KB blocks. The leaf level tables have 64 entries which each provide individual permissions for 16 four-byte words.

We next examine different formats for the entries in the trie table.

3.2.1 Permission Vector Entries

A simple format for a trie table entry is a vector of permission values, where each leaf entry has 16 two-bit values indicating the permissions for each of 16 words, as shown in Figure 3-4. User segments are represented with the tuple < base addr, length, permissions>. Addresses and lengths are given in bytes unless otherwise noted. The user segment `<0xFFC, 0x50, RW>` is broken up into three permission vectors, the latter two of which are shown in the figure. We say an address range *owns* a permissions table entry if looking up any address in the range finds that entry. For example, in Figure 3-4, `0x1000`–`0x103F` owns the first permission vector entry shown.

Upper level trie table entries could simply be pointers to lower level tables, but to reduce space and run-time overhead for large user segments, we allow an upper level entry to hold either a pointer to the next level table or a permissions vector for sub-blocks (Figure 3-5). Permission vector entries in the upper levels contain only eight sub-blocks because the upper bit is used to indicate whether the entry is a pointer or a permissions vector. For example, each mid-level permissions vector entry can represent individual permissions for the eight 512 B blocks within the 4 KB block mapped by this entry.

User segments can be any number of words starting at any word-aligned address, but these might be broken into one or more table segments, according to the size and alignment

```
PERM_ENTRY trie_table_lookup(addr_t addr) {
  PERM_ENTRY e = root[addr >> 22];
  if(is_tbl_ptr(e)) {
    PERM_TABLE* mid = e<<2;
    e = mid[(addr >> 12) & 0x3FF];
    if(is_tbl_ptr(e)) {
      PERM_TABLE* leaf = e<<2;
      e = leaf[(addr >> 6) & 0x3F];
    }
  }
  return e;
}
```

Figure 3-3: Pseudo-code for the trie table lookup algorithm. The table is indexed with an address and returns a permissions table entry. The base of the root table is held in a dedicated CPU register. The implementation of `is_tbl_ptr` depends on the encoding of the permission entries.

restrictions of the trie table. This process is completely transparent to the user, but it has performance implications. For instance, if a 9 word segment is granted read-write permissions, then two bottom-level entries must be updated. If the user segment started on an 8-word aligned address, then the permissions information for 8 words would be in the first table segment, and information for one word would be in the second table segment. If an address indexes into the second segment, the loaded entry only has permissions for one word, not the entire nine word user segment. In the next section we introduce a new entry type to address this limitation.

3.2.2 Run-length encoded entries

A permissions value boundary is where word N has one permissions value and word $N+1$ has a different value. Most user segments are longer than a single word, so any run of N words is likely to have fewer than N permission value boundaries. We can take advantage of this property by run-length encoding permissions values in a table entry, to make a *run-length encoded* (RLE) entry.

The sorted segment table demonstrated a compact encoding for abutting segments—only base and permissions are needed because the length of one segment is implicit in the base of the next. A run-length encoded entry uses the same technique to increase the encoding density of an individual trie table entry. The result looks like a small version of a sorted segment table, where the base for one entry implicitly defines the length of the previous entry.

Figure 3-6 shows the bit encoding for a run-length encoded entry, which can represent up to four table segments crossing the address range that owns the entry. As with the SST, start offsets and permissions are given for each segment, allowing length (for the first three entries) to be implicit in the starting offset of the next segment. We chose four table segments because our measurements of the malloc behavior of the programs in Table 5.1 showed that the size of heap allocated objects is usually greater than 16 bytes (4 words). A single run-length encoded entry can represent 4 adjacent 4-word protection regions.

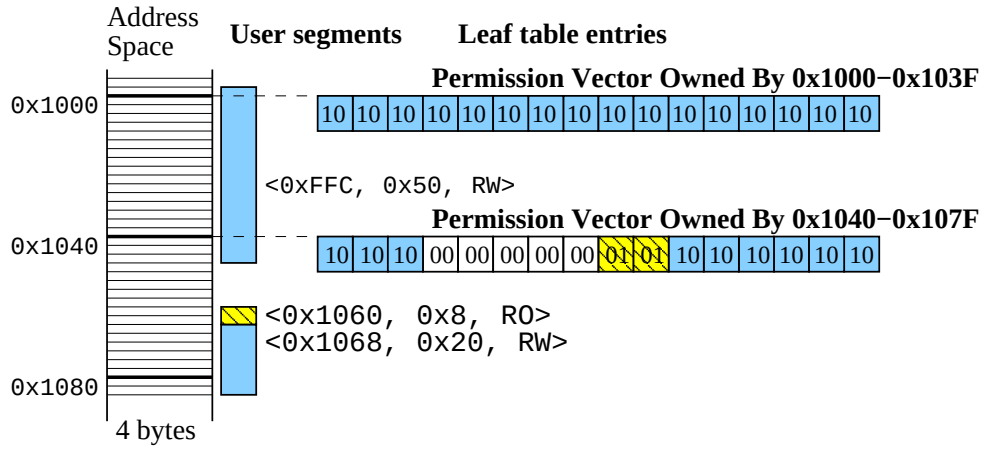
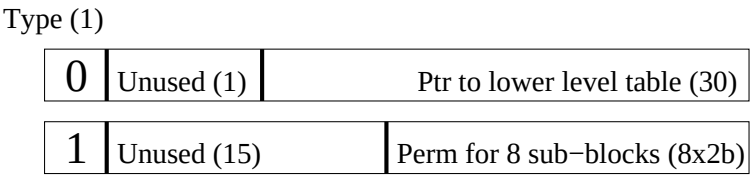


Figure 3-4: A trie table entry consisting of a permissions vector. User segments are broken up into individual word permissions.




```
bool is_tbl_ptr(PERM_ENTRY e){return(e>>31)==0;}
```

Figure 3-5: The bit allocation for upper level entries in the permissions vector trie table, and the implementation of the function used in trie_table_lookup.

Run-length encoded entries represent permissions for a larger region of memory than just the 16 words (or 16 sub-blocks at the upper table levels) that own it. The `first` segment has an offset which represent its start point as the number of sub-blocks (0–31) *before* the base address of the entry's owning range. Segments `mid0` and `mid1` must begin and end within the entry's 16 sub-blocks. The `last` segment can start at any sub-block in the entry except the first (a zero offset means the `last` segment starts at the end address of the entry) and it has an explicit length that extends up to 31 sub-blocks from the end of the entry's owning range. The largest span for an entry is 79 sub-blocks (31 before, 16 in, 32 after).

The example in Figure 3-4 illustrates the potential benefit of storing information for words beyond the owning address range. If the entry owned by `0x1000`–`0x103F` could provide permissions information for memory at `0x1040`, then we might not have to load the entry owned by `0x1040`.

Figure 3-7 shows a small example of run-length encoded entry use. Segments within an SST entry are labeled using a <base, length, permission> tuple. Lengths shown in parentheses are represented implicitly as a difference in the base offsets of neighboring table segments. The entry owned by `0x1000-0x103F` has segment information going back to `0xFFC`, and going forward to `0x104C`. The address range `0x1000-0x103F` is split across the first and last run-length encoded table segments. The middle entries are not needed (both specify an offset of 15, which give them an implicit length of zero).

Run-length encoded entries can contain overlapping address ranges, which complicates

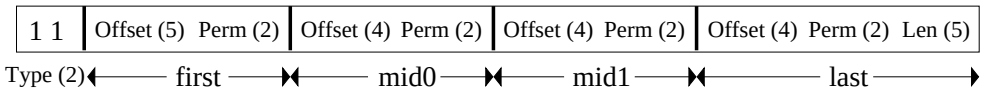


Figure 3-6: The bit allocation for a run-length encoded permission table entry.

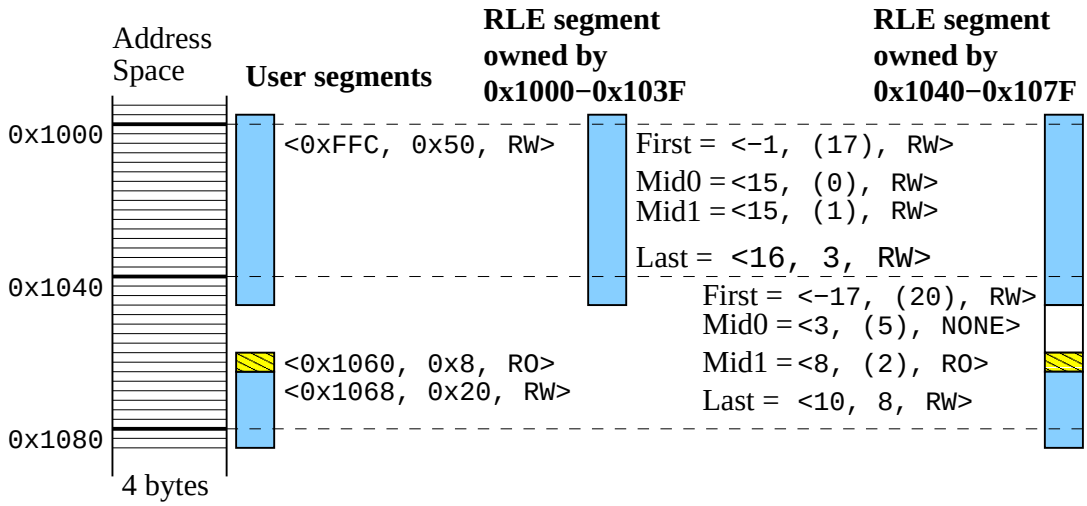


Figure 3-7: An example of segment representation for run-length encoded entries.

table updates. When a user of MMP changes the entry for one range, MMP must update any other entries that overlap with that range. For example, if a user of MMP frees part of the user segment starting at `0xFFC` by protecting a segment as `<0x1040, 0xC, NONE>`, MMP must read and write the entries for both `0x1000`–`0x103F` and `0x1040`–`0x107F` even though the segment written by the user does not overlap the address range `0x1000`–`0x103F`.

One restriction we impose to simplify table update is that an upper level entry cannot overlap with memory owned by an entry which is a pointer to a lower level table. Without this restriction, MMP must search surrounding entries at every level in the table to update any possibly overlapping entries. The cost of these extra table accesses is not justified by the benefit of the overlap, so such overlap is disallowed.

We can design an efficient trie table using run-length encoded entries as our primary entry type. The run-length encoded format reserves the top two bits to encode an entry's type tag; Table 3.2 shows the four possible types of entry. The upper tables can contain pointers to lower level tables. Any level can have a run-length encoded entry, and any level can contain a pointer to a vector of 16 permissions. This restriction is necessary because run-length encoded entries can represent only up to four abutting segments. If a region contains more than four abutting segments, we represent the permissions using a permission vector held in a separate word of storage, and pointed to by the entry. So four memory references are sufficient to find the permissions for any address using the trie with run-length encoded entries. Finally, the format specifies a pointer to a record that has a run-length encoded entry and additional information. We use this extended record to implement translation as discussed in Section 10.2.

Type	
00	Pointer to next level table.
11	Run-length encoded entry (4 segments spanning 79 sub-blocks).
01	Pointer to permission vector (16x2b).
10	Pointer to run-length encoded+ (e.g., translation (6x32b)).

```
bool is_tbl_ptr(PERM_ENTRY e){return(e>>30)==0;}
```

Table 3.2: The different types of trie table entries, and the implementation of the function used in trie_table_lookup. *Type* is the type code. Leaf tables do not have type `00` pointers.

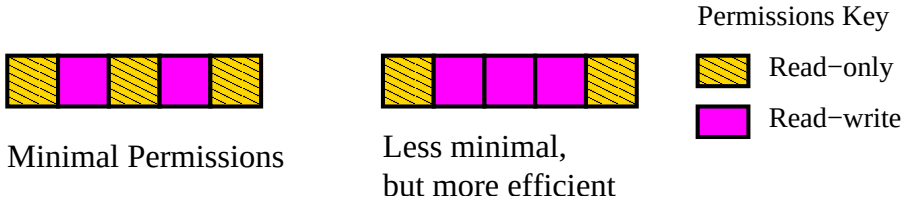


Figure 3-8: Permissions distribution for five words. On the left, write permission is only granted on the word which needs it. On the right, the entire read-write area has been coalesced.

Balancing minimal permissions and performance with RLEs

MMP provides word-level memory protection so a software service can have permissions on the smallest subset of memory necessary for the service's function. However, run-length encoded entries can only represent a limited number of permissions regions for a given block of memory. Consider the permissions distribution of the five words in Figure 3-8. On the left, the words have the minimum level of permissions needed for the program to function properly. However, there are four permissions value boundaries, which is the limit of what can be represented with a run-length encoded entry.

In the picture on the right, the middle word has been given read-write permissions even though it does not need to be (and should not be) written. There are now only two permissions value boundaries. Sometimes being generous with permissions will allow the MMP system to be more efficient by allowing a single table entry to hold permissions about a larger region of memory.

3.3 Gate tables

Up to this point, the permissions table have held two bits of permissions information per word. This is sufficient to protect data, and to distinguish code from data, but it is insufficient to represent gate permissions. MMP uses gates to allow the hardware to guarantee that cross-domain control transfer to a domain only happens at a location approved by that domain. MMP gates are simpler than those present in the x86, IA-64, or PA-RISC because they do not cause a stack switch.

Table 3.3 shows the two gate types in the MMP system, the *switch gate* and the *return gate*. These gate permissions values are placed on instructions by the MMP system at the behest of the user. For instance, during a module's initialization, the module's code would place gates on its exported functions. The initialization code places a switch gate on the first

Gate type	Gate data
switch	destination PD-ID
return	

Table 3.3: Gate types and their associated data. Switch gates are used to change protection domains, and they specify their destination protection domain. Return gates need no additional data.

instruction of an exported function, and a return gate on the control flow instruction that returns from the procedure. For most compilers, the procedure return is the last instruction of the function.

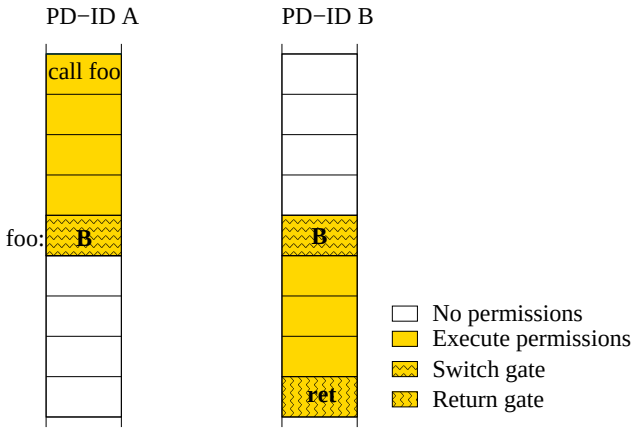


Figure 3-9: How gate permissions are placed on instructions for cross-domain calling. In the example, domain B exports the routine named foo.

When an exported function is called, MMP uses the switch gate from the caller's domain and the return gate from the callee's domain. Consider the example in Figure 3-9. For a thread executing in domain A to call the routine `foo` implemented by domain B, the thread transfers control to the first instruction of `foo`, probably using the function call instruction for the given architecture. As seen in the figure, the first instruction of `foo` has the switch gate permission value, so the processor will initiate a domain switch to the domain specified by the gate, in this case domain B. When the routine returns, the return gate notifies the processor that it must switch back to domain A.

The switch gate is placed on the first instruction of the function, not on the call instruction (as is done in the x86 and PA-RISC architectures), which is why we call it a switch gate instead of a call gate. Placing the permissions on the first instruction of the routine means that call sites don't have to be identified when a function is exported, and a single instruction can call exported and non-exported routines. Like the x86 and PA-RISC, the domain switch happens before the first instruction of the exported routine is executed.

There are many ways to represent gate permissions in the permissions tables. If they are considered permissions values in addition to the values in Table 3.1, then the permissions table would need 3-bit entries to represent the 6 possible values. However, switch gates require additional storage for the destination protection domain identifier. Architectures with byte-aligned call and return instructions (e.g., x86) would require two extra bits to encode the gate location because the permissions tables hold entries for words, not bytes.

Instead, we store gate information in its own table. The number of gates, even for a

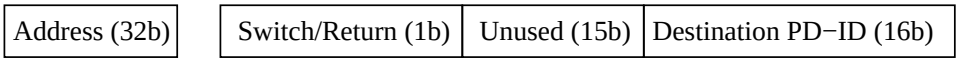


Figure 3-10: The format of an entry in the gate permission table.

large system, is low (less than 1,000 in Mondrix), because modules tend to have many more internal functions than exported entry points. We store the gates in an open hash table. The format of an entry is shown in Figure 3-10. It consists of a byte address, which is the location of the gate instruction (for architectures that specify 32-bit instructions, this would be a word address). The second word of the entry specifies the gate type, and if it is a switch gate, the destination protection domain.

We considered using bits in the upper level entries to classify regions as code or data. The two bits of permissions data would be interpreted as in Table 3.1 for data pages, and as no-access, execute-read, switch-gate, return-gate for code pages. Each page could have a single destination PD-ID, also specified in the upper level entry (this would require a different encoding from the one presented in Table 3.2).

3.4 Possible table optimizations

The space of possible permissions encoding is large, so we present possible improvements that would reduce the size of the permissions table, or would require fewer memory accesses to find a particular entry.

3.4.1 Extension to 64-bits addresses

We can extend the multi-level MMP table to accommodate a 64-bit address, using many of the same techniques that were used to extend page tables to a wider address space [THK95]. Because MMP supports arbitrary-sized regions, it requires forward-mapped indexing. A forward-mapped scheme requires five levels of table lookup, where the top 3 level tables have 4K entries, and the last two levels have 2K entries. To make lookups faster, the hardware or software that reads the permissions table hashes the top 42 bits of the address. This hash is used as an index into a table which has either permission entries or pointers to the lowest two level tables used by the five table lookup path. The hardware or software that reads the permissions table also updates the hash table whenever a lookup fails, with the entry retrieved from searching the five level tables from the root. The space consumption for this strategy will be larger than the 32-bit case, but we believe the time consumption could be tuned to be close to the 32-bit case.

3.4.2 Sharing permission tables

Permissions tables are potentially large because they contain permissions information for an entire address space. The trie table organization allows domains that have identical permissions values for a region of their address space to share permissions tables, subject to alignment constraints.

For instance, in Mondrix, several domains have the same permissions on the page that contains a task's user area. This page contains several permissions regions at odd locations

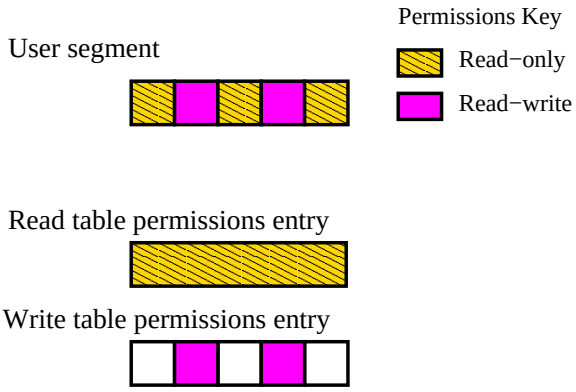


Figure 3-11: How independent tables for independent permissions values can yield efficient entries.

and sizes, so it is contained in a leaf level table. Instead of replicating that table for each domain that uses it, it can be filled in once, and pointed to by several domains. This optimization requires additional bookkeeping to determine when the table is no longer being used.

3.4.3 Alternate permissions encodings

In this section we briefly discuss alternative formats for table entries.

64-bit permissions entries

The run-length encoded entries in Section 3.2.2 occupy 32 bits. A 64-bit entry could be superior, because it is easier to amortize more status bits in larger table entries. More status bits allow specialized entry types which can increase lookup efficiency. For instance, upper level entries could be redesigned to handle a small number of regions that are not multiples of their sub-block length. Easing the alignment restriction for upper level sub-blocks would bring more upper level entries into use, and reduce the average number of memory accesses required for permissions lookups. The entries can also take advantage of a wide data path to memory.

Multiple 1-bit tables

The design we present encodes 4 permission values using 2 bits. We could use 3 tables, each with 1 bit of permissions for each word. One table would be for execute permissions, one for write, and one for read. If a word has no permissions in all tables, it cannot be accessed. This encoding is very efficient at representing read permissions on large regions which alternate read and read-write permissions. Figure 3-11 shows an example user segment, and how it would be represented in the read permission table and the write permission table. With the 2-bit encoding, there are 4 permissions value transitions. With multiple 1-bit tables, there are 4 permissions value transitions in the write permission table, but there are no transitions in the read permission table, possibly allowing the read table to use an entry which covers more memory (e.g., an entry in an upper level table).

Since read-only permissions are often an acceptable alternative to no permissions, a system with an independent read permissions table might be more efficient in terms of time and space. By eliminating permission transitions between read-only and read-write, more upper level entries might be used.

Global table (unified entries)

The MMP design specifies that each protection domain has its own permissions table. As the number of protection domains grows, the amount of storage dedicated to protection tables might grow as well. MMP reduces the space consumed by multiple domains by efficiently supporting large regions, and by allowing some sharing of permission table between domains.

If a system has many domains, but domains tend to have permissions on disjoint memory regions, a single table could be used for all domains, where the entry for an address indicates which domains have access permissions. The global table would track which domain has permissions on each word of memory. Each entry in such a table would contain some number of protection domain identifiers, and the permissions each domain has on a given range of memory.

A global table makes global entries (entries that are the same for every domain) easy to implement. The tension with a global table is that for space efficiency, the entries should be small. But a small entry may only accommodate a limited number of protection domain identifiers and their permissions. For efficient lookup, every domain that is sharing a word of memory should be listed in the entry, which requires larger entries. Directories for cache-coherence have the same tension between exact sharing information and entry size, and the trade-off has been addressed by such diverse mechanisms as: sharing lists, coarse bitvectors, and adaptive schemes where sharing lists degrade to coarse bitvectors.

Chapter 4

MMP Hardware

The MMP hardware must check every instruction fetch, and the address of every load and store instruction. In order to make the permissions check efficient, the processor caches the permissions information on chip. The cache should minimize trips to permission table memory, it should not slow the processor's critical path, and it should not unduly increase the CPU's energy consumption.

This chapter presents a design to achieve all of these objectives. The design uses two levels of on-chip cache, the first level (the protection lookaside buffer or PLB) eliminates trips to the permissions table in memory, the second level (sidecar registers) avoids the energy cost of searches in the PLB.

Intuitively, MMP can be implemented efficiently because it caches a range of permissions in a single entry. Just as a page of words shares a translation value, allowing it to be efficiently cached by a TLB, MMP can cache permissions more effectively by representing permissions for many words in a single entry. The performance of TLBs and page tables has kept pace with the rigorous performance demands of current computing systems, so we can expect MMP do to the same.

4.1 Lookaside Buffers

Figure 4-1 is a more detailed version of Figure 1-2, which shows all of the major components of the Mondriaan system, including the support for switch and return gates. The gate PLB is refilled by a hardware state machine that reads the gate table in a way similar to the hashed page table search in the PowerPC architecture [IBM02]. The gate table is structured as an open hash table, that the hardware indexes when a program counter value is not present in the gate PLB.

4.1.1 Protection Lookaside Buffer (PLB)

The protection lookaside buffer (PLB) caches protection table entries just as a TLB caches page table entries. The PLB hardware uses a conventional ternary content addressable memory (CAM) structure to hold address tags that have a varying number of significant bits (as with variable page size TLBs [KH92]). The PLB tags have to be somewhat wider

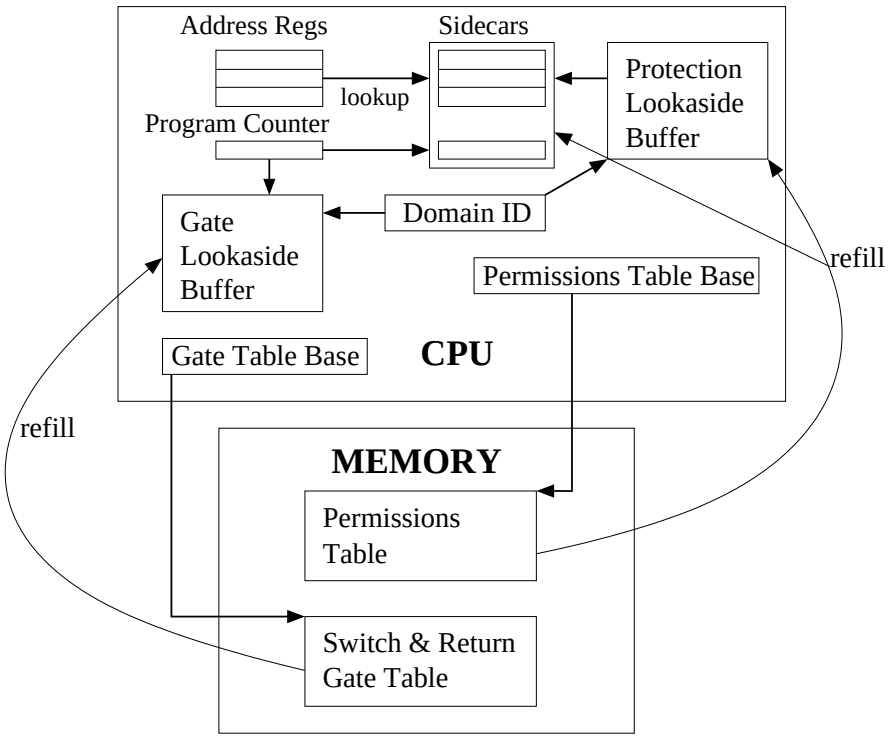


Figure 4-1: The major components of the Mondriaan memory protection system, with support for switch and return gates.

than a TLB because they support finer-grain addressing (26 tag bits for our example design). Entries are also tagged with protection domain identifiers (PD-IDs).

The ternary tags stored in the PLB entry can contain low-order "don't care" address bits to allow the tag to match address ranges. For example, the tag `0x10XX`, where `XX` are don't care bits, will match any address from `0x1000`–`0x10FF`. On a PLB refill, the tag is set to match addresses within the largest naturally aligned power-of-two sized block for which the entry has complete permissions information. Referring to the example in Figure 3-7, a reference to `0x1000` will pull in the entry for the block `0x1000`–`0x103F` and the PLB tag will match any address in that range. A reference to `0x1040` will bring in the entry for the block `0x1040`–`0x107F`, but this entry can be stored with a tag that matches the range `0x1000`–`0x107F` because it has complete information for that naturally aligned power-of-two sized block. This technique increases effective PLB capacity by allowing a single PLB entry to cache permissions for a larger range of addresses.

When a program changes the permissions for a region in the permissions tables, the MMP system must flush any out-of-date PLB entries. Permissions modification occurs much more frequently than page table modifications in a virtual memory system. To avoid excessive PLB flushing, we use a ternary search key for the CAM tags to invalidate potentially stale entries in one cycle. The ternary search key has some number of low order "don't care" bits, to match all PLB entries within the smallest naturally aligned power-of-two sized block that completely encloses the region we are modifying (this is a conservative scheme that may invalidate unmodified entries that happen to lie in this range). A similar scheme is used to avoid having two tags hit simultaneously in the PLB CAM structure. On a PLB refill, the hardware or software doing the refill must first invalidate all PLB entries

that overlap with the range of the entry being fetched before writing the new entry into the PLB. The invalidation process takes a single cycle using low-order "don't care" bits to match a power-of-two sized address range.

There is an algorithm that allows two ternary CAMs to represent an arbitrary address range [PS03]. The CAMs are searched in parallel to allow a single cycle lookup. If the PLB used this two CAM structure, it could index an exact range, not the largest enclosed power-of-two sized region.

4.1.2 Gate protections lookaside buffer (GPLB)

Just as gate permissions have their own table (Section 3.3), they also have their own lookaside buffer, the GPLB. Mondrix uses a large set-associative lookaside buffer rather than a small, fully associative buffer. The gate PLB could be a simple CAM with an address and PD-ID tag and the gate type and data, but experiments with Mondrix show a less than 10% hit rate with a 64-entry CAM. For the Mondrix evaluation in Chapter 8 uses a 512-entry, 4-way set associative cache for gate permissions.

4.2 Sidecar registers

Lookups in the PLB's associative store can consume a significant fraction of on-chip energy, because associative lookups broadcast the key value to all storage cells. Arm's low-power StrongARM architecture dissipates 17% of its on-chip energy in TLB lookups [MWA+96] (the StrongARM has fully-associative instruction and data TLBs). MMP has an additional, optional, level of cache, called *sidecar registers* which eliminate the energy cost of accessing the PLB.

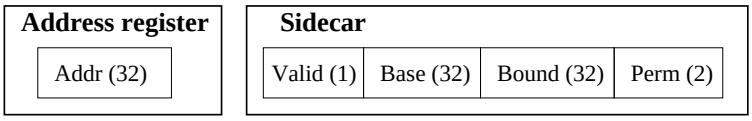


Figure 4-2: The layout of an address register with its sidecar register.

Each architectural address register in the machine has an associated sidecar register, which holds information for one table segment as depicted in Figure 4-2. The program counter has its own sidecar used for instruction fetches.

On a PLB miss, the hardware or software miss handler looks up the demand address in the permissions table, and writes the permissions table entry for that address into the PLB. It also writes the table segment into the sidecar of the address register that the processor used to calculate the effective address of the memory load or store. All fields of the table segment descriptor are represented in uncompressed form in the address sidecar to facilitate fast checking of base, bounds and permissions. The base and bounds information is constructed by combinational logic based on the demand address, the level of the table from which the entry was read, and the entry itself.

For each subsequent load or store using that base register, the processor compares the effective address against the base and bounds in the register's sidecar. If the address lies within the range, the processor uses the sidecar permissions value. If the range check fails or the sidecar is invalid, the processor searches the PLB for the correct permissions

information. The PLB might miss, causing the refill mechanism to access the permissions table in memory.

Sidecars also increase the permissions hit rate by caching an entire table segment. The PLB can often index only part of the permission table entry because the PLB's index range must be a naturally aligned power-of-two sized block. For example, in Figure 3-7 a reference to `0x1040` will load the segment `<0xFFC, 0x50, RW>` into the register sidecar. If that register is used to access location `0xFFC` we will have a permissions check hit from the sidecar. Sending `0xFFC` to the PLB will result in a permissions check miss because it only indexes the range `0x1000`–`0x107F`.

To guarantee consistency, MMP invalidates all sidecars when any protections are changed so the sidecars never cache stale permissions values. The processor invalidates all sidecars on protection domain switches. If sidecars had PD-ID tags (which might be useful for the stack pointer, a global data pointer or cross-domain argument pointers), invalidation on domain switch would not be necessary. But the processor can refill sidecars rapidly from the PLB, so domain ID tags are not part of the design.

Register sidecar information is like a capability in that it has protection and range information, but it is not managed like a capability because it is ephemeral and not user visible. Sidecars are similar to the resolved address registers in the IBM System/38 [HSH81], where an address such as the base of an array would be translated, cached and then reused to access successive array elements.

4.3 Cross-domain calling

This chapter provides additional detail on how cross-domain function calls are protected in MMP, explaining the cross-domain call stack, the handling of interrupts, and how to pass arguments.

4.3.1 Gate requirements

When a thread executes an instruction with a switch or return gate permission, the architecture must perform certain operations. We list the abstract operations, and then show how to realize these operations efficiently, and with minimum modification, to an existing architecture.

A cross-domain switch has the following requirements:

- Store and protect the call information (e.g., the return address and the caller's protection domain identifier).
- Make the callee's protection domain the current domain, and start execution at the called instruction.

A cross-domain return has the following requirements:

- Look up the saved return address and caller's protection domain identifier.
- Verify that control is returning to the proper return address.
- Make the caller's domain the current domain, and start execution at the saved return address.

If any of the checks fail, the call or return does not succeed, the hardware generates a fault, and restarts execution in the memory fault handler (implemented as part of the supervisor, discussed in Section 6.1).

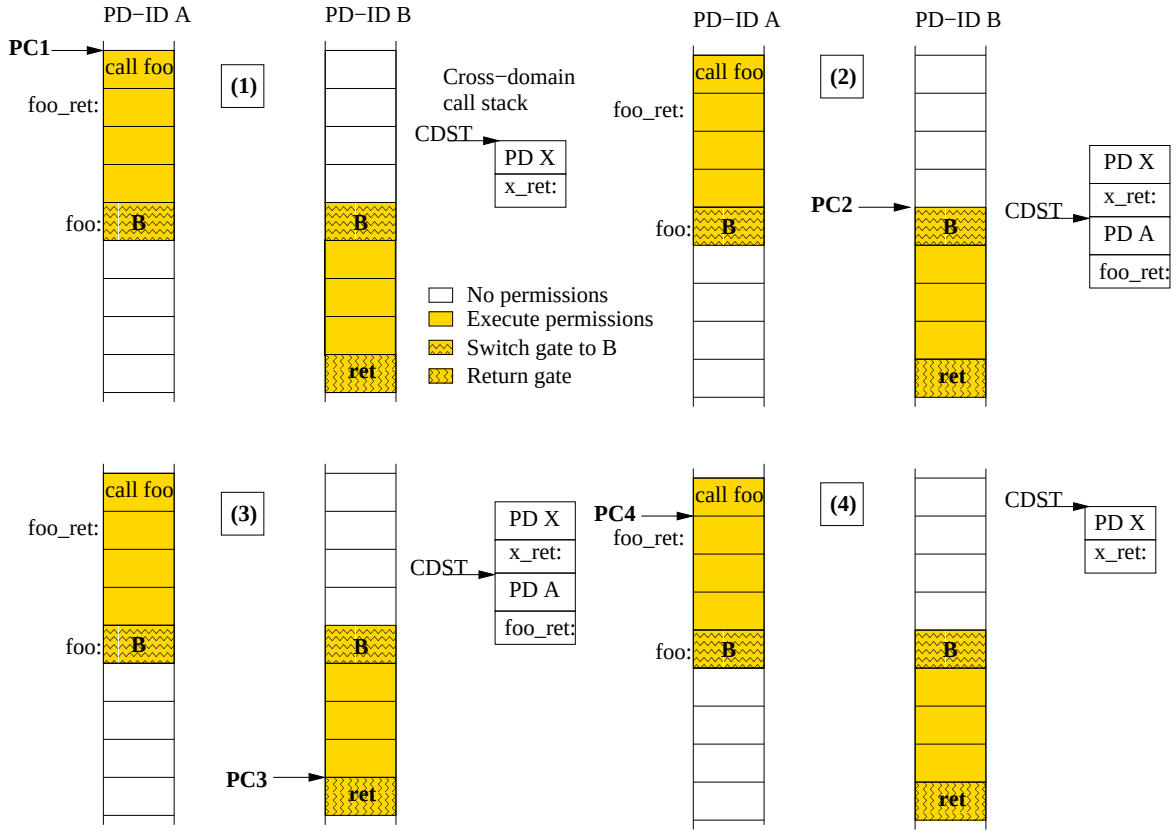


Figure 4-3: How MMP is used for cross-domain calling. `PC1` – `PC4` indicate the program counters during four points in a cross-domain call. The program starts in domain A in the portion of the figure labeled **(1)**. The `CDST` (cross-domain stack top) register points to the record for the call to A. Domain A was called by some domain (which we call X), at some program point (whose return address we denote by the label x_ret). The processor implementation of the call instruction finds the switch gate in domain A (in the portion of the figure labeled **(1)**), and switches to domain B to execute the first word of `foo` (labeled **(2)**). The destination domain of the switch gate (B) is part of the gate. Execution of the switch gate causes the processor to store the return address, and the PD-ID of the caller protection domain on the cross-domain call stack. On execution of the return gate (labeled **(3)**) the processor verifies that it is returning to the caller's protection domain at the proper address (labeled **(4)**).

4.3.2 Gate implementation

The *cross-domain call stack* is an ordinary area of memory that the hardware can write, but most software can only read. For operating systems that maintain a kernel stack per process, each process has its own cross-domain call stack. We add an additional hardware register, the `CDST`, or cross-domain stack top register, which holds the memory address of

the current record on the top of the cross-domain call stack. The memory supervisor (the software part of the MMP system) saves and restores this register on a context switch. It relies on the OS to notify it about scheduling events.

To implement a switch gate, the processor checks the target PC for every control flow instruction. If a switch gate is present on the target instruction, the call state is saved on the cross-domain call stack, and the CPU state is changed to a new protection domain (the PD-ID is changed, along with the base pointer to the domain's permissions table, as seen in Figure 1-2).

In parallel with instruction fetch, the processor checks for a return gate on a return instruction. If found, the processor reads the cross-domain call stack to find the saved return address. It checks that the return address for the return instruction matches the saved address. Then, it changes the protection domain to the stored value, and resumes execution at the return address.

On a cross-domain switch, the processor increments `CDST` by the size of a PD-ID and a return address, and then it writes the current PD-ID and return address into the new location specified by `CDST`. On a cross-domain return, the processor reads the call record from the location in `CDST`, and decrements the register.

Our gate implementation is applicable to RISC and CISC architectures and a variety of function call instructions, because it only involves a permissions check and pushing a call record on the cross-domain call stack.

4.3.3 Cross-domain call example

A thread calling and returning from another protection domain is shown in Figure 4-3. Any control flow instruction can initiate a cross-domain call, though it will usually be a standard subroutine call instruction. `PC1` shows the program counter about to execute the call instruction. After the pipeline stage where the control-flow instruction determines the new PC, the processor checks for a gate on that location. In the example, the thread executing in Domain A finds the switch gate on `foo` within domain A. Before the processor executes the first instruction of `foo`, it changes protection domain from A to B. So `PC2` shows the processor executing the first instruction of `foo` in domain B. When the processor executes the last instruction of `foo` (`PC3`), it changes the domain changes back to A, and sets the PC to the instruction after the call (`PC4`).

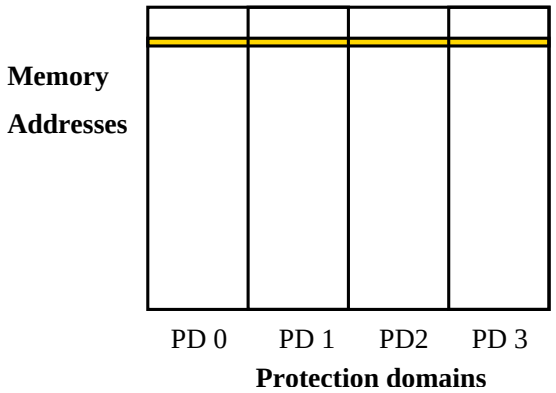


Figure 4-4: How the same code (e..g, interrupt stubs) can be mapped into every domain.

The hardware must store both the caller's return address and protection domain identifier. A set of domains can share code (depicted in Figure 4-4), so the protection domain identifier is needed to determines the return domain—the return address is insufficient. Sharing code among multiple protection domains is useful; the kernel requires all modules to share interrupt handling code (see Section 7.5.1).

The processor executes return gates in the callee's domain, which causes problems if a domain calls a function that it 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 establish the state needed for a cross-domain return. 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. However, this decision has the unfortunate consequence of approximately doubling the number of cross-domain calls (see the data in Section 8.5). Fortunately, cross-domain calls that don't change PD-ID require less micro-architectural work because the processor does not need to invalidate the sidecar registers, and the control registers don't change.

The presence of a switch gate to domain B within B itself is shown in Figure 4-3. Switch gates are read in the caller's domain, so B's switch gate is only read by calls originating within B itself.

4.4 Hardware implementation issues

MMP protection checks must be efficient for today's processor designs, and the extra mechanism needed for MMP should require few changes to the processor implementation. We consider adapting an in-order and an out-of-order issue processor to implement MMP.

4.4.1 In-order pipeline implementation

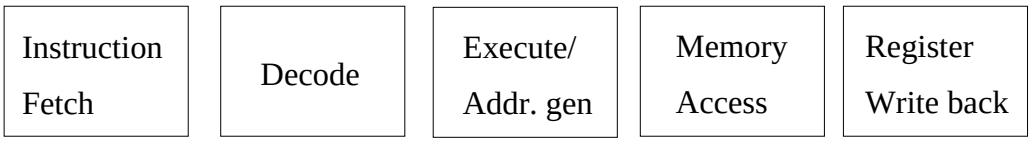


Figure 4-5: An in-order, five-stage pipeline.

Figure 4-5 shows the stages of a five-stage pipeline for an in-order processor. During the address generation phase of the pipeline (Execute/Addr. gen in Figure 4-5), the processor checks the address sidecar corresponding to the base register specified in the instruction. If the sidecar does not provide the permissions information for the current address (either because it is invalid or because it contains protection information for an address range that does not include the effective address), the processor looks up the address in the PLB, and writes the result into the sidecar. The processor checks for a fault before the register write back stage of the pipeline.

The entire pipeline is flushed on a cross-domain call, just as it is flushed on a system call. For short, in-order pipelines, the performance overhead of flushing the pipeline is low.

4.4.2 Out-of-order pipeline implementation

For an out-of-order (OOO) issue machine, the MMP permissions check need only complete just before instruction graduation, allowing the latency of the check to overlap with instruction execution. Overlapping the permissions check with instruction execution means the permissions check adds no latency to an instruction in the common case, and it indicates that MMP will not increase processor cycle time because it is out of the critical path to instruction completion.

An OOO issue processor contains support for speculatively loading and storing data. They will execute loads that occur after a predicted branch, and will roll back execution if the branch was mispredicted. They also contains a store buffer for speculative stores. MMP permissions checks use the mechanisms already present in an OOO issue processor. The processor may use speculative load data before permissions are verified. Similarly, store data is only committed out of the speculative store buffer once the processor verifies that the address being stored has write permissions.

Sidecars are primarily an energy optimization, so their presence might be more compelling in a simpler machine, rather than a complex out-of-order issue machine which already consumes a lot of on-chip energy. If sidecars are used in an out-of-order processor, they are physically located by the load/store unit and only need as many read ports as the number of simultaneous load and store instructions supported. The processor uses the architectural register number (not the renamed physical register number) to index into the sidecar register file. A misspeculated load can bring in the wrong table segment, but this causes subsequent accesses only to miss in the sidecar, and to retrieve their permissions from the PLB. Protection breaches are not possible, even if sidecar updates are not done in program order. The information in the sidecar must always be valid, but it need not be relevant to the current access.

OOO issue processors could incur large performance penalties if cross-domain calls caused a serialization of instruction execution Domain switches can be made considerably faster by associating protection domain ID values with each instruction in the pipeline, eliminating the need to flush the pipeline on a cross-domain call. Multiple protection domains in MMP can share the reorder buffer using almost the same mechanism that multiple addressing contexts use to share the reorder buffer in a processor that supports simultaneous multi-threading (SMT) [TEL95] (called "hyper-threading" by Intel). In both cases, instructions from different contexts share the execution resources of the machine. In MMP, each thread has its own PD-ID, in SMT, each thread has its own ASID.

A cross-domain call should be higher performance than a context switch. Context switches change the address translation context, and any miss to the address translation cache (TLB) stalls the execution of that instruction (and its dependents). Cross-domain calls change the protection context, and misses to the protection translation cache (PLB) do not stall the execution of an instruction, because the instruction speculates that the protection check will succeed. The processor should be able to overlap the execution of more instructions from two different protection contexts than from two different addressing contexts.

4.4.3 Mixing mapped and pinned memory

Most operating systems place their code and much of their data in physical memory. This memory is not paged, it is "pinned," because it never resides on disk. Most operating systems map their code and data into every user process. However, to conserve physical memory, many operating systems (e.g., Linux and Solaris) support kernel operation with a mix of pinned and mapped memory. The bottom half of the kernel is pinned, while the top half, usually containing the user page tables, is mapped.

Mixing mapped and pinned memory poses a difficulty for MMP, because a single virtual address can refer to different memory locations at different times during execution. So long as kernel mapped memory tracks the user context (as it does when it contains the user page tables), MMP can make the address space identifier (ASID) part of the PD-ID, so each mapped context has a different PD-ID. If the use of mapped memory in the kernel is not so stylized, the mapping code must inform the MMP system when and how mappings change, creating an additional bookkeeping burden on the MMP system. To simplify OOO superscalar processor design, instructions are tagged only with the PD-ID, and all instructions in the reorder buffer share an ASID.

User address spaces consist solely of mapped memory, so this problem does not arise for them.

4.4.4 The problem with inlining code

MMP protection domains assume contiguous code all belongs to a single domain. Inlining frustrates that assumption. Compiler frameworks which make heavy use of inlining (e.g., [ASG97]) would need a new approach, or might frustrate MMP entirely. A fine-grained interleaving of instructions from different protection domains is difficult to represent. Gates at all transition instructions would require too many bits, because a gate entry contains the instruction address for the gate, and the switch gate contains the destination PD-ID. For instance, on a 32-bit RISC machine, if every 5th instruction came from a different protection domain, the gate permissions overhead would be 50%, and the hit rate of the GPLB would be poor.

Systems that inline code at runtime (e.g., using a just-in-time (JIT) compiler) are compatible with MMP. The domain in which new code is inserted is called the target domain. So long as a JIT can identify the data accessed by the code it inlines, it simply marks that data as shared by the target domain, and marks the code as part of the target domain. MMP provides mechanism for data sharing, but can not represent finely-interleaved code from different protection domains.

4.4.5 Approaches for multi-processors

Permissions tables have the same coherence issues as page tables in a multi-processor system. If a thread modifies the permissions tables, any processor which might be caching the data must invalidate its sidecar registers and the relevant portions of the PLB. If the PLB maintains inclusion with the second level cache, it will be notified if an entry it is caching is evicted from the cache. This reduces the effectiveness of the PLB because any permissions table entry it caches must be resident in the second level cache, but it provides a way to keep PLB's coherent by piggybacking on existing coherence hardware.

Chapter 5

MMP Evaluation for User Programs

We have presented a complete MMP design. To validate our design, we built a simulator of the MMP hardware, and measured a variety of user-level applications as they used MMP for different purposes. The purpose of the experiments is to evaluate permissions table designs, and verify that our MMP system can efficiently provide real applications with useful memory protection services.

We present a detailed performance analysis of the tables. When an application uses MMP for coarse-grained protection, MMP's space consumption is small (<1% of the application memory usage), and it adds few additional memory references (<1% of the application memory references). When an application uses fine-grained memory protection, MMP's space cost is under 9% and the number of extra memory references it adds is under 8%.

5.1 Evaluation Methodology

Fine-grain memory protection is useful, but comes at a cost in both space and time. The permission tables occupy additional memory and accessing them generates additional memory traffic. The time and space overheads depend on three issues: where the compiler and memory allocator place data in memory, how the programmer protects that data, and how the program accesses that data.

We evaluated both C and Java programs. C programs were compiled with `gcc` version `egcs-1.0.3a` for a 32-bit MIPS target using -O3 optimization and static linking to generate an ELF binary. The malloc from the `newlib` library was used. The linker and malloc libraries were used unmodified. The results would be better if the linker was modified to align and pad program sections, and if malloc was modified to try to align addresses and to place its internal management state away from the returned memory. The Java programs were compiled for a MIPS target using MIT's FLEX Java-to-native compiler [RAB+03]. FLEX's output was also linked with the `newlib` malloc library. The garbage collector in FLEX was disabled for all of our runs to put a heavier load on the memory allocator.

A big challenge in evaluating MMP is trying to predict how programmers would take advantage of word-granularity protection. In this evaluation, we considered two extreme cases. In the first case, we assumed light use of the protection facilities. We ran the programs in a single protection domain with the standard protection regions for Unix processes: read-only program text, read-only data, read-write data, and stack. This level of protection is what is provided by most current operating systems. In the second case, we assume that every object allocated by malloc is in a separate user segment and that the surrounding words are inaccessible because they hold malloc internal state. The malloc implementation is one protection domain, and the program is another. However, gate permissions were not used for these experiments because there are only a few entry points to malloc, so any caching scheme would work well.

5.2 Benchmark overview and methodology

To gather data on how programs access data, we chose a mix of benchmarks that were both memory reference and memory allocation intensive. Table 5.1 lists the benchmarks used and their reference properties. Benchmark names prefixed with a "j-" are Java programs. Benchmarks `crafty`, `gcc`, `twolf` and `vpr` are from SPEC 2000, and `vortex` is from SPEC 95. The `_tr` suffix indicates the training input, and `_test` suffix indicates the test input. Names prefixed "o-" are from the Olden [Car96] benchmark suite. Names prefixed with "m-" are from the Mediabench benchmark suite. Table 5.1 includes the number of memory references per table update. Only `malloc`, `realloc`, and `free` update the permissions table, and the results show a wide variation in how frequently objects are created and deleted.

The programs were run on a MIPS simulator modified to trace data memory references as well as calls to `malloc`, `realloc`, and `free`. We considered only data references because the instruction reference stream remains inside a single text segment for these codes, but we put the protection information for the text segment in the permissions table. These traces were fed to our model implementations of the SST and the trie which keep track of size of the tables, and the memory accesses needed to search and update the tables. The implementation also models all invalidates of sidecars and PLB required for consistency with table updates, and to prevent multiple hits in the PLB after refills.

We measure space overhead by measuring the space occupied by the protection tables and dividing it by the space being used by the application for both program text and data at the end of a program run. We determine the space used by the application by querying every word in memory to see if it has valid permissions. As a result, the space between malloced regions is not counted as active memory even though it contributes to the address range consumed by malloc and to the protection table overhead. The stack starts at 64 KB and is grown in 256 KB increments. Each call to `brk` returns 1 MB.

We approximate the effect on runtime by measuring the number of additional memory references required to read and write the permission tables. We report overhead as the number of additional references divided by the number of memory references made by the application program. The performance impact of these additional memory references varies greatly with the processor implementation. An implementation with hardware PLB refill and a speculative execution model should experience lower performance overheads because these additional accesses are not latency critical. A system with software PLB refill and a simple pipeline should have higher relative time overhead. In addition to counting additional memory references, we also fed address traces containing the table accesses to a

Benchmark	Refs $\cdot 10^6$	Segments	Refs/Update	Cs
crafty_test	3,088	96	64,327,162	6
gcc_tr	1,684	20,796	161,944	26
twolf_train	11,537	938,844	24,576	8
vpr_test	506	6,274	161,191	6
vortex_tr	1,291	211,630	12,200	16
j-compress	561	6,430	174,554	14
j-db	109	249,104	876	12
j-jack	402	1,622,330	496	34
j-jess	245	215,460	2,275	10
j-raytrace	1,596	1,243,052	2,567	20
m-jpeg_dec	1	58	45,785	6
m-mpeg2_dec	30	46	1,307,794	6
o-em3d	608	131,598	9,240	22
o-health	142	846,514	336	14

Table 5.1: The reference behavior of benchmarks. The *Refs* column is total number of loads and stores in millions. The *Segments* column is the number of segments written to the table during the fine-grained experiments (which is twice the number of calls to `malloc` since each call effectively creates two segments). The next column is the average number of memory references between updates to the permissions table. *Cs* is the number of segments when running with coarse-grained protection. There are a variable number of coarse-grained segments because each call to `brk` (extending the heap), and each OS extension of the stack creates a new segment.

cache simulator to measure the increase in miss rate caused by the table lookups.

For the permissions caching hierarchy, we placed register sidecars on all 32 integer registers. The results used either a 64-entry or 128-entry PLB with 4 entries reserved for the supervisor and a random replacement policy. We do not model the supervisor code in our experiments, and so we report just the number of PLB entries available to the application (60 or 124).

5.3 Coarse-Grained Protection Results

Table 5.2 shows the space and time overhead results for the coarse-grained protection model. We present the results only for the trie with run-length encoded entries and a 60-entry PLB. We contrast the overheads of the permissions table with a model of a page table and TLB. The page table and TLB provides four regions, executable code, read-only data, read-write data, and stack. The linker pads the program sections to page boundaries. The MMP system uses the true lengths of the program sections, not their rounded values. It also distinguishes read-write static data from bss data. Using the real length of the program sections is significant because their length is almost always an odd number of words, which requires leaf level tables, and therefore more space. Also, MMP uses a new region for each additional chunk of memory returned by the OS in response to a `brk` system call in a new segment, and each chunk of new stack memory demand mapped by the OS. Stack and heap extension account for the variable number of coarse-grained segments (column *Cs* in

Benchmark	Trie RLE 60 PLB			PAGE+TLB		
	X-ref	Space	l/k	X-ref	Space	l/k
crafty_test	0.56%	0.41%	2.1	2.59%	0.15%	2
gcc_tr	0.01%	0.08%	2.0	0.17%	0.03%	2
twolf_train	0.00%	0.31%	2.0	0.76%	0.11%	2
vpr_test	0.00%	0.62%	2.6	0.00%	0.22%	2
vortex_tr	0.02%	0.10%	2.0	0.77%	0.04%	2
j-compress	0.00%	0.11%	2.1	2.16%	0.04%	2
j-db	0.32%	0.17%	2.0	0.98%	0.06%	2
j-jack	0.00%	0.04%	2.2	0.04%	0.02%	2
j-jess	0.06%	0.18%	2.1	0.59%	0.06%	2
j-raytrace	0.00%	0.07%	2.2	0.01%	0.03%	2
m-jpeg_dec	0.27%	0.61%	2.8	0.12%	0.22%	2
m-mpeg2_dec	0.01%	0.61%	2.3	0.01%	0.22%	2
o-em3d	0.00%	0.07%	2.1	0.02%	0.03%	2
o-health	0.02%	0.12%	2.1	0.07%	0.05%	2

Table 5.2: The extra memory references *X-ref* and extra storage space *Space* required for a run-length encoded permissions table and 60 entry PLB used to protect coarse-grain program regions. We compare to a traditional page table with a 60 entry TLB. The *l/k* column gives the average number of loads required for a table lookup, which is a measure of how much the mid level entries are used for permission information.

Table 5.2).

The overheads are small in both space and time for both MMP and a TLB system. The trie space overhead is bigger than the page table overhead, but it is less than 0.7% for all of the benchmarks, compared with the application's memory usage without MMP. The trie uses additional space because of the leaf level tables that accommodate program sections whose start or end addresses are not divisible by 256 B. If the program segments were aligned, and grew in aligned quantities, the trie and page table would consume the same space.

The trie adds fewer than 0.6% extra memory references compared to the application running without MMP, and requires fewer table accesses than the page table for every benchmark except Mediabench's `mpeg2`. The `mpeg2` run is so short that writes to set up the permission table make up a large part of the table accesses. The advantage of the trie is the reach of its mid-level run-length encoded entries. These entries are for 4 KB of address space, but they can contain information for a 20 KB region. A conventional page table entry has information only for 4 KB ranges. For instance, `compress`, a benchmark known to have poor TLB performance, mallocs a 134 KB hash table which is accessed uniformly. This table requires 33 TLB entries to map, but only requires 8 entries in the worst case for the PLB. The number of loads per lookup is close to 2 indicating that mid-level entries are heavily used.

We also simulated an SST with a 60-entry PLB. This configuration performs much better that either of the previous schemes, with both time and space overheads below 0.01% on all benchmarks, compared to the application running without MMP. The ability of the SST table segments to represent large regions results in extremely low PLB miss rates. Given the number of segments listed in Table 5.1, all of the benchmark's coarse-grained segments

probably fit into the PLB at the same time. Because there are so few coarse grain segments, the lookup and table update overhead is small.

These results show that the overhead for MMP word-level protection is less than 1%, compared with an application that does not use MMP, when MMP is used for large segments.

5.4 Fine-Grained Protection Results

We model the use of fine-grain protection with a standard implementation of malloc which puts 4–8 bytes of header before each allocated block. We remove permissions on the malloc headers and only enable program access to the allocated block. We view this as an extreme case, as a protected subsystem will typically only export a subset of all the data it accesses, not its entire address space.

Table 5.3 shows the results for the fine-grain protection workloads. While the SST organization performs well for some programs, its time and space overhead balloons on other programs. For `o-health` the space overhead reaches 44%. The binary search lookup has a heavy, but variable, time cost, which can more than double the number of memory references. For `j-jack`, it averages 20.8 loads per table lookup, but for `mpeg2` it is only 4.8. Because SST must copy half the table on an update on average, updates also cause significant additional memory traffic. But SST does have significantly lower space and time overheads than the trie table for some applications like `gcc` and `crafty`. The `gcc` code mallocs a moderate number of 4,072 byte regions for use with its own internal memory manager. This odd size means the trie table must use leaf tables which have limited reach in the PLB, while the SST represents these segments in their entirety. More flexible mid-level table entries (see Section 3.4.3 might be able to represent these odd-sized regions.

All trie table organizations take almost exactly the same space and so their space overhead is reported together in one column. The space overhead for the trie table is less than 9% for all permission entry types. Because the run-length encoded format must use an escape to a bitvector when the number of permission regions in one entry is greater than 4, it can require a little more space than the permission vector format. Five of the benchmarks required permission vector escapes, but only two required more than 30 escapes. The `health` benchmark required 4,037 pointers to permissions vectors in the leaf entries, and `j-jess` 332. Although is not likely to represent real program behavior [Zil01], `health` provides a stress test for our system because it allocates many small segments.

We garbage collect trie permission tables when they become unused. This keeps memory usage close to the overhead of the leaf tables, which is $1/16 = 6.25\%$ because information for 16 words is held in a 32-bit permissions table entry. Some overheads are higher than 6.25% because of non-leaf tables. Each table has a counter with the number of active entries. When this counter reaches zero, the table can be garbage collected. The reads and writes to update this counter are included in the memory reference overhead.

The run-length encoded organization is clearly superior to the permission vector format (compare columns **vec 60 PLB** to **RLE 60 PLB**). Every benchmark performs better and the highest time overhead (`vpr`) is more than halved, dropping from 19.4% to 7.5%. Lookups dominate the additional memory accesses, as expected. `jpeg` and `mpeg` from Mediabench are small programs that don't run for long so updating the tables is a noticeable fraction of table memory references for these benchmarks. `j-jack` has high update overhead because it performs many small allocations with little activity in between (from Table 5.1 it does

Benchmark	SST 60 PLB				Space	vec 60 PLB			RLE 60 PLB			RLE 124 PLB		
	Space	X-ref	upd	ld/lk		X-ref	upd	ld/lk	X-ref	upd	ld/lk	X-ref	upd	ld/lk
crafty_test	0.0%	0.0%	49%	7.4	0.6%	3.2%	1%	2.1	0.6%	1%	2.1	0.0%	1%	2.1
gcc_tr	0.2%	0.7%	36%	13.4	4.0%	3.6%	4%	2.8	1.5%	13%	2.9	1.0%	19%	2.9
twolf_tr	22.2%	141.0%	63%	16.5	6.6%	10.6%	1%	3.0	7.5%	1%	3.0	6.3%	1%	3.0
vpr_test	0.1%	0.7%	96%	11.2	4.5%	19.4%	1%	2.9	7.5%	1%	2.9	1.4%	1%	2.9
vortex_tr	0.8%	105.0%	95%	16.0	4.5%	4.3%	3%	2.8	2.4%	7%	2.8	1.2%	13%	2.9
j-compress	0.2%	0.0%	54%	12.8	0.4%	3.1%	1%	2.2	0.1%	9%	2.4	0.0%	59%	2.7
j-db	16.3%	69.1%	5%	19.2	4.9%	7.4%	7%	2.9	6.4%	8%	3.0	5.6%	9%	3.0
j-jack	23.5%	20.0%	31%	20.8	6.9%	4.8%	18%	2.9	3.0%	27%	2.9	2.1%	39%	2.9
j-jess	12.7%	22.0%	7%	18.7	4.8%	3.4%	6%	2.9	2.6%	8%	2.9	2.1%	10%	3.0
j-raytrace	30.5%	10.1%	11%	21.4	6.8%	1.1%	12%	3.0	1.0%	14%	3.0	0.8%	17%	3.0
m-jpeg_dec	0.0%	0.0%	75%	4.8	6.3%	3.1%	9%	2.9	0.5%	64%	3.0	0.4%	86%	3.0
m-mpeg2_dec	0.0%	0.0%	71%	5.2	7.2%	0.1%	18%	2.8	0.0%	71%	2.8	0.0%	85%	2.7
o-em3d	3.2%	16.2%	2%	18.7	6.5%	2.6%	8%	3.0	2.1%	9%	3.0	1.7%	12%	3.0
o-health	44.0%	75.3%	12%	20.0	8.3%	7.6%	13%	3.0	6.1%	17%	3.0	5.7%	18%	3.0

Table 5.3: Comparison of time and space overheads with inaccessible words before and after every malloced region. The *Space* column is the size of the permissions table as a percentage of the application's active memory. The last three organizations are all trie tables and all occupy about the same space. The *X-Ref* column is the number of permissions table memory accesses as a percentage of the application's memory references. The *upd* column indicates the percentage of table memory accesses that were performed during table update. The remainder of the references are made during table lookup. The *ld/lk* column gives the average number of loads required for a table lookup.

less than 500 memory reference in between table updates). When we increase the number of available PLB entries to 124 (column **RLE 124 PLB**), the worst case memory reference overhead drops to 6.3%, with some benchmarks, like `vpr`, benefiting greatly.

The format of the trie table entries is important because some encodings can represent information about a larger address range. The PLB contains permissions information for a fixed amount of memory known as its *reach*. The reach of a TLB without super-page support is the number of entries times the page size. The reach of the PLB depends on the format of the permission table entries. Bitvector entries always have the same reach, but run-length encoded entries can have a larger reach. While their reach is variable, on average it is higher than the bitvector entries. Run-length encoded entries are superior to bitvectors because a larger PLB reach means fewer PLB misses, which reduces costly data cache misses to the permissions table.

In the trie organization, leaf table entries are undesirable because their entries have a limited range, and so many of them must be resident to cover a large address range. The SST's direct representation of the user segment is an advantage because user segments don't get broken up when they are placed into the permission table. If they fit application allocation behavior, flexible mid-level table entries (see Section 3.4.3), where a large memory range is broken into a small number of protection regions on odd boundaries, would cover permissions for more memory than a leaf table entry.

5.5 Memory Hierarchy Performance

	Coarse	**Fine**		
Benchmark	**SCar**	**SCar**	**PLB**	**SCar Elim**
crafty_test	28.5%	28.5%	0.3%	1.0%
gcc_tr	9.4%	11.4%	0.4%	3.3%
twolf_train	15.5%	17.8%	2.5%	1.7%
vpr_test	37.3%	42.5%	2.6%	7.2%
vortex_tr	12.4%	15.0%	0.8%	2.4%
j-compress	5.6%	22.9%	0.0%	11.4%
j-db	14.2%	18.4%	2.0%	2.6%
j-jack	7.3%	9.8%	0.8%	1.9%
j-jess	8.3%	16.6%	0.8%	1.1%
j-raytrace	0.8%	2.5%	0.3%	0.6%
m-jpeg_dec	7.0%	13.2%	0.1%	10.9%
m-mpeg2_dec	7.4%	7.4%	0.0%	4.2%
o-em3d	12.8%	13.1%	0.7%	7.0%
o-health	5.6%	8.6%	1.7%	3.8%

Table 5.4: Measurements of miss rates for a trie table with run-length encoded entries and a 60 entry PLB. *SCar* is the sidecar miss rate. *PLB* is the global PLB miss rate (PLB misses/total references). *SCar Elim* is the number of references to the permissions table that were eliminated by the use of sidecar registers for the fine-grained protection workload. For coarse-grained protection, the PLB miss rates were close to zero on all benchmarks and so are not shown here.

Table 5.4 shows the performance of the permissions caching hierarchy including the sidecar miss rate and the PLB global miss rate for the fine-grained protection workload. The sidecar registers normally capture 80–90% of all address accesses, while the PLB captures over 97% of all address accesses for all benchmarks.

We also show the percentage reduction in references to the permissions tables as a result of using sidecar registers. The principal motivation for using sidecars is to reduce traffic to the PLB, but there is a significant performance gain also (more than 10% for two benchmarks) because some sidecar hits would be PLB misses, as explained in Section 4.2.

As another indirect measure of performance cost, we measured the increase in miss rate caused by the additional permissions table accesses. The results for a typical L1 cache (16 KB) and a typical L2 cache (1 MB) are shown in Figure 5.5. Both caches are 4-way set associative. For the L1 cache, at most an additional 0.25% was added to the miss rate, and for the L2 cache, at most 0.14% was added to the global miss rate but most applications experienced no difference in L2 miss rates.

The miss rates do not increase more than the 8% increase in memory references, so the references to the permission table exhibit a locality commensurate with the applications. In general the increase in miss rate is small, and even negative in two cases. Set associative caches that use LRU replacement often do not implement an optimal cache replacement policy [SA93]. Small perturbations in a reference stream can lower the miss rate by improving the replacement policy.

Benchmark	**16 KB, 4-way**			**1 MB, 4-way**		
	App	**MMP**	**Δ**	**App**	**MMP**	**Δ**
crafty_test	1.86%	1.87%	0.01%	0.01%	0.01%	0.00%
gcc_tr	4.25%	4.30%	0.06%	0.22%	0.22%	0.00%
twolf_train	2.82%	3.04%	0.21%	0.00%	0.00%	-0.00%
vpr_test	3.37%	3.62%	0.25%	0.00%	0.00%	0.00%
vortex_tr	0.71%	0.72%	0.02%	0.10%	0.10%	0.00%
j-compress	2.82%	2.82%	0.00%	0.12%	0.12%	0.00%
j-db	2.25%	2.39%	0.14%	0.50%	0.53%	0.03%
j-jack	0.54%	0.55%	0.01%	0.24%	0.24%	0.01%
j-jess	0.84%	0.86%	0.02%	0.07%	0.07%	0.00%
j-raytrace	0.22%	0.23%	0.00%	0.03%	0.03%	0.00%
m-jpeg_dec	0.43%	0.43%	-0.00%	0.09%	0.09%	0.00%
m-mpeg2_dec	0.20%	0.20%	-0.00%	0.04%	0.04%	0.00%
o-em3d	0.42%	0.42%	0.01%	0.19%	0.20%	0.00%
o-health	2.44%	2.58%	0.14%	2.41%	2.55%	0.14%

Table 5.5: *App* is the cache miss rate of the application benchmark, while *MMP* is the combined cache miss rate for the references of the benchmark and the MMP protection structures. Δ is their difference. A trie table was used with run-length encoded entries and a 60 entry PLB. This table holds results from two experiments, differing only in cache size—16 KB and 1 MB. The cache was a 4-way set-associative with 32-byte lines. -0.00 means the miss rate decreased slightly. Reference streams were simulated for a maximum of 2 billion references.

Chapter 6

MMP Memory Supervisor

Like a traditional memory management unit, MMP hardware needs system software to present a useful abstraction to the user. The software side of MMP is the memory supervisor, which provides software support for MMP's permissions abstractions.

This chapter presents an abstract description of the supervisor, but one informed by Mondrix. In particular, the supervisor design tries to be minimal, to make slipping it "under" an existing operating system as easy as possible.

The supervisor manages the permissions tables, and controls how domains are created, named, and destroyed. It establishes policy on how domains can share memory, and provides an interface to memory allocators that allows allocators to give permission on memory blocks to client domains. Finally, it provides the new abstraction of group protection domains, which manage permissions on multiple disjoint memory regions.

One important design principle for the supervisor is that it cannot trust the parameters passed to it by other system software. It guards against buggy code, so it must verify pointers, and check that requested operations are legal. Its relationship to the rest of the kernel is analogous to the kernel's relationship to user programs.

This chapter first explains the concepts that the memory supervisor manages. Then it shows the protection domain structure for the supervisor and a modular application, followed by a comprehensive overview of the supervisor. Then Section 6.4 reviews the supervisor's application programming interface (API). Section 6.5 describes the memory supervisor's policies. Section 6.6 discusses how the supervisor works with a memory allocation service to broker permissions efficiently without replicating the work that the allocator does. Finally, Section 6.7 discusses two important data structures that the supervisor uses.

6.1 Memory supervisor concepts

An understanding of the memory supervisor begins with understanding the abstractions the supervisor implements and manages. *Access permissions* describe what a domain can do with memory, i.e., the kinds of permissions discussed up to this point, e.g., read-write, or return gate. A protection domain's permissions table describes its memory access permissions. We call memory *accessible* if there is some way for a program to access it without causing a fault (i.e., by reading, writing, or executing it). *Export permissions* describe how

a domain can export permissions to another domain. If a domain only has read-only export permissions on a region, it cannot export read-write permissions on it to another domain. Memory is *shared* when it is accessible by more than one domain.

Each word in memory is *owned* by exactly one protection domain. A protection domain has rights on memory it owns, for instance it can assign access permissions on it.

A *group protection domain* is a collection of memory segments, each with a specified permission. Code executing in a protection domain creates a group protection domain and *exports* memory segments with a specified permission value to the group domain. A protection domain gains the permissions specified by the segments in a group protection domain when it *adds* the group domain. (The domain which creates a group domain can also add it.) Group domains are useful for data that has a single function, but the memory segments that serve that function change with time. In Mondrix, read-write kernel stacks are in a group protection domain. The memory regions in this group domain change as processes are created and destroyed. The memory supervisor regulates which protection domains can add a group.

6.2 Using the supervisor for a modular application

The MMP hardware checks every load, store and instruction fetch for proper permissions. Protection domain zero (PD 0) is exempt from this checking so it can read and write the permissions tables themselves. The memory supervisor runs in PD 0 so it can maintain the protection tables; but the high privilege granted to code in PD 0 provides motivation to keep that code as small as possible.

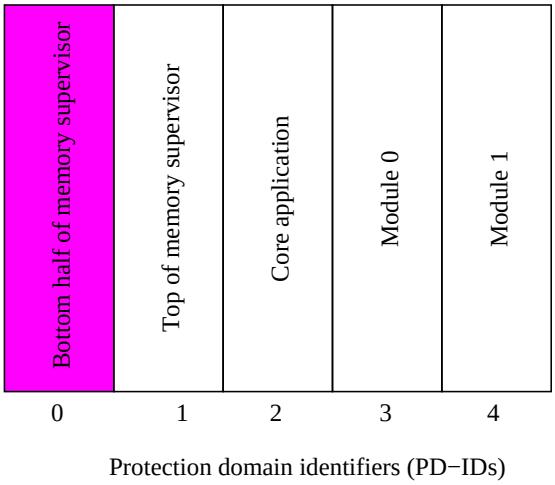


Figure 6-1: Structuring a generic, modular application to use multiple domains. The bottom half of the memory supervisor is resident in protection domain 0 (PD 0), which has unmediated access to all of memory. The top half is in PD 1. The core of the application is in PD 2, and modules are loaded in PD 3, and its successors.

Figure 6-1 shows how the supervisor is used by a generic, modular application. To minimize the amount of code in PD 0, the memory supervisor is split into two parts. The bottom half writes permissions tables, updates the MMP hardware, and accepts memory faults, providing a hardware-independent view of the permissions table to the users of the

MMP hardware. These low level operations must be exempt from the hardware checking done by the MMP hardware. Fortunately this list of low-level operations is short, and the operations are simple.

All of the supervisor software entry points and most of the supervisor code is in PD 1. The core application (such as an OS kernel) goes in the next domain with modules or other subsystems loaded into subsequent domains. By loading its modules into different domains, the application isolates its modules, forcing them to make their memory sharing patterns explicit. MMP enforces the modular boundaries that are already present in software systems.

6.3 Memory supervisor overview

The memory supervisor serves several functions; we discuss each in turn to give an intuitive understanding of its operation. All of these functions are implemented by the code in PD 1, except for the code which changes access permissions by writing protection tables. That code is in PD 0.

The memory supervisor serves several functions:

1. It provides a hardware independent abstraction of the permissions tables.
2. It checks requests for memory access permissions, determining if a protection domain has the right to change the access permission for a region of memory.
3. It implements memory ownership.
4. It implements, and checks requests for memory export permissions.
5. It implements creation and deletion of protection domains.
6. It tracks memory sharing across domains, supporting permissions revocation for dynamically allocated memory or protection domain deletion.
7. It implements group protection domains.

The first task of the memory supervisor is to manage the protection tables to provide a higher-level, hardware-independent interface for permissions to the users of the MMP hardware. Only the low-level supervisor code knows about table entry encoding, the rest of the system simply manages protection information on arbitrary runs of contiguous words.

When users request additional access permissions, (e.g., by putting a switch or return gate on an instruction), or when they downgrade their permissions (e.g., making a data buffer read-only), the memory supervisor checks their request. Our design does not allow users to manipulate their own permissions tables—the supervisor manages all permission requests, allowing it to enforce a policy for memory use.

The supervisor implements memory ownership. Every address space is divided into non-overlapping regions, where each region is owned by exactly one protection domain. When a protection domain owns a memory region, it is responsible for how that memory region is used in the system. Specifically, an owner can set arbitrary access permissions on memory that it owns (just like the owner of a file in the Unix file system can add or take away file permissions at will). A protection domain can also export arbitrary permissions on memory that it owns to other domains. While memory access permissions are intended for

fine-grained use, memory ownership is intended to be more coarse-grained. Coarse-grained ownership reduces the implementation complexity of the supervisor, without compromising performance. The supervisor maintains a sorted list (the SST described in Section 3.1) of memory regions and their owners. This data structure has a small number of large entries that change infrequently, making an SST an appropriate data structure.

The supervisor implements export memory permissions, which are the rights a domain has to grant other domains access permissions to its memory. Ownership conveys unlimited export permissions, but non-owning domains can have limited export permissions. For instance, an owning domain can give another domain (call it domain X) read-write access permissions on a buffer, but only read-only export permissions. Domain X can read and write the buffer, but cannot grant read-write permissions on the buffer to domain Y.

The supervisor manages the creation and deletion of protection domains. The creation of a domain consists of specifying the memory that it owns, and the deletion of a domain consists of finding a new owner for memory owned by the deleted domain. The supervisor maintains a tree of protection domains to track the parental relationships between domains. Most of the work for domain creation and deletion is to enforce policy. For example, the Mondrix supervisor only allows a domain to create another domain using memory it owns, and when a domain is deleted, it revokes permissions from all domains on memory owned by the deleted domain.

The supervisor tracks memory regions that are shared among domains. When access or export permissions on these regions must be revoked, (e.g., because a program frees a piece of dynamically allocated memory, or because a protection domain is deleted), the supervisor revokes permissions only from domains which can access the memory. The supervisor could revoke permissions on shared memory from every domain, but permissions revocation is done as part of freeing dynamically allocated memory, which is a frequent event in the kernel (and in most applications), and so it must be done efficiently. Writing permissions tables is an expensive operation, so only writing the tables of the domains that can access the memory being freed is an important optimization. Having the supervisor track memory sharing (as opposed to having the owning domain do it) insures that memory is properly reclaimed, preventing resource leakage.

The supervisor implements group protection domains, which are collections of memory regions, with specified permissions, that are united by a common use. For instance, inodes are a kernel data structure that record 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 of inodes that a module can add 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.

6.4 Memory supervisor API

This section reviews the supervisor API (printed in Appendix A) for domain maintenance, and memory permissions manipulation, dynamic memory allocation, naming protection domains, and group protection domains.

6.4.1 Protection domain creation

The supervisor provides the function `pd_subdivide(struct mmp_req*)`, which allows a protection domain to divide itself, creating a new domain. It takes a list of requests, each specifying a memory region and permission value, and it returns the protection domain ID of the new domain in which the memory is placed. The `steal` flag indicates if the region is supposed to be owned by the new protection domain. If `steal` is true, the memory region is no longer owned by the original owner, but is owned by the new protection domain.

6.4.2 Protection domain deletion

Protection domains are created hierarchically, and they are destroyed hierarchically. The supervisor tracks the entire protection domain hierarchy, allowing parents to call `mmp_free_pd(pd_id, recursive)` on their children. If the recursive flag is true, all of the deleted protection domain's children are also deleted. Otherwise, they are re-parented to the closest surviving parent remaining in the tree. The hierarchical structure mimics the Unix process structure because we thought it would be appropriate for tracking domains. However, domains could have any relationship to each other, the system just needs to be able to assign a new owner for the memory owned by a deleted protection domain.

6.4.3 Changing memory permissions

The basic operation for setting permissions is provided by the `mmp_mprot(ptr, len, prot, pd_id)` call. The `ptr` parameter is the start address of the region, and the `len` parameter is the length. The start address is rounded down to the nearest 4-byte address, and the length has the low 2 bits of the address added to it, and it is then rounded up to the nearest multiple of 4. The `prot` parameter is the protection value, which can be one of `PROT_NONE`, `PROT_READ`, `PROT_READ|PROT_WRITE`, and `PROT_EXECUTE|PROT_READ`. Finally, the `pd_id` parameter identifies the target protection domain. This function is analogous to `mprotect`, but works at word granularity, and it has the additional, `pd_id` parameter, which specifies the target domain for the permissions.

This function lets a protection domain set permissions on its memory, and lets it export permissions to another protection domain. For example, the `ide-disk` domain makes this call:

```
mmp_mprot(&idedisk_driver, sizeof(ide_driver_t), PROT_READ, kern_pd);
```

which exports a structure called `idedisk_driver` read-only to the kernel domain.

6.4.4 Setting gate permissions

Gate permissions values are stored in a separate structure from the standard permissions table (see Section 3.3), and the interface for setting gate permissions is different as well. The supervisor exports the routine, `mmp_func_gate(func, pd_id)`, which takes a function pointer and a protection domain. A program should set gate permissions only on the first and last instruction of a routine, and the supervisor enforces this policy. The supervisor verifies that the function pointer has execute permissions and that it is a function entry point (using symbol information, see Section 7.2.1). If the `pd_id` parameter is equal to the owner of the function pointer, then a switch gate is set on the first instruction of the routine, and a return gate is set on the last instruction. If the given protection domain is not equal

to the owner of the function, then a switch gate to the owner's protection domain is set on the first instruction of the routine. The supervisor determines the owner of the routine by searching its list of owners for the entire address space.

For example, the real-time clock domain sets gate permissions during initialization like this:
`mmp_func_gate( &rtc_open, rtc_pd );`
`mmp_func_gate( &rtc_open, kern_pd );`.
These calls make the function `rtc_open` a valid entry point for the kernel, and allows the rtc domain to call the routine itself via a cross-domain call.

The memory supervisor, and occasionally a domain itself, might want to know the permissions a domain has on a piece of memory. This information is provided by the function: `mmp_get_prot(ptr, pd_id, prot*)`. This function reads the protection value on the word at location `ptr`, and stores it in the variable pointed to by the `prot` parameter.

6.4.5 Dynamic memory allocation

A memory allocator's allocation function (e.g., `kmalloc` or `vmalloc` in Mondrix) calls `mmp_mem_alloc(ptr, len)`, and the allocator's free function (e.g., `kfree` or `vfree`) calls `mmp_mem_free(ptr, len)`, passing them the location and length of the memory block being allocated or freed. These functions are similar to `mmp_mprot`, but the target protection domain is set to the caller's caller. The allocator domain calls these functions using a cross-domain call to the supervisor, so the caller on the cross-domain call stack is the allocator's domain; the caller's caller is the requesting domain. The supervisor obtains the requesting protection domain identifier by reading the cross-domain call stack, because hardware guarantees its correctness.

The protection value granted to the requesting domain is implicit, and set by the memory supervisor. For memory the allocating domain owns, the supervisor sets the permissions on allocation to read-write (justification is in Section 6.6.1). For memory the domain does not own, the supervisor sets the permissions on allocation to the access permissions value the allocator domain has on the memory. The supervisor reads the allocator's domain to determine its access permissions. The supervisor revokes permissions when memory is freed. The `mmp_mem_free` call requires a length parameter, which is provided by the allocator.

6.4.6 Naming domains

There are several ways to name protection domains. The domain identifier of the calling domain is available to the caller by reading the cross-domain call stack, which lets a service safely and efficiently determine who called it.

The supervisor maintains a sorted list of owners for regions of memory. This allows code addresses, and addresses of static data to be attributed to protection domains via the function `mmp_code_to_pd(const void*)`, which takes a pointer and returns the protection domain that owns the pointer. This routine performs a binary search of all owned memory regions, so it cannot be called in circumstances that require a constant time operation.

Finally, the supervisor maintains a collection of protection domain identifiers whose names correspond to a certain module. For example, `rtc_pd` is the protection domain for the `rtc.o` module (the real-time clock). If the value of this variable is zero, the module

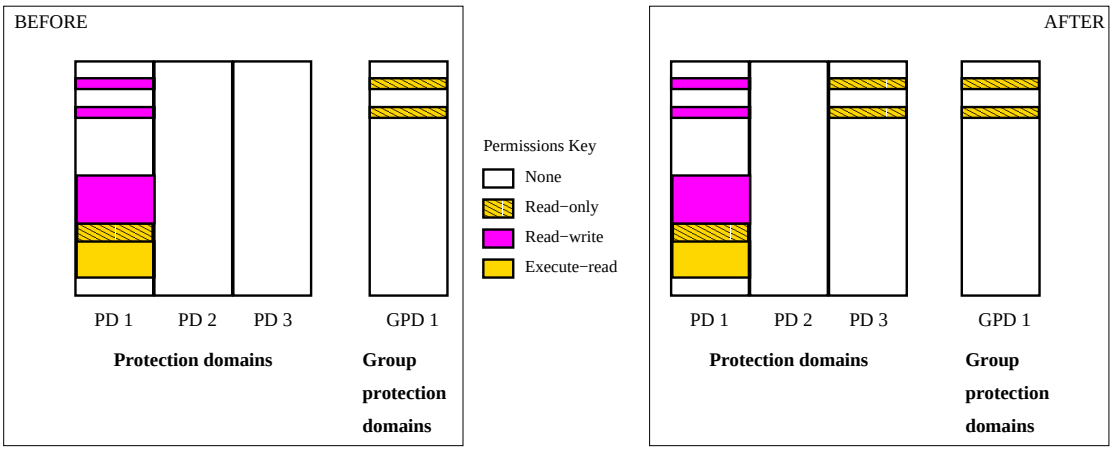


Figure 6-2: An example of a group protection domain. In this case, protection domain 1 has read-write permissions on two regions of memory. It exports read-only permissions on both to group protection domain 1. On the right of the legend, protection domain 3 has added group protection domain 1, so it gains read-only permissions to the two pieces of memory.

has not been loaded. This simple mechanism is convenient, and is similar to the current practice in the kernel of naming domains by a string containing their name.

6.4.7 Group protection domains

Group protection domains are virtual protection domains that contain a group of memory segments, each with an associated permission. The memory supervisor tracks which memory locations belong to a group, and regulates which modules can join a group. Group protection domains are created by calling `mmp_gpd_create(const char* name, int nregions)`, which takes the name of the group and an estimate of the number of memory segments the group will contain and returns a group protection domain identifier (gpd_id). The program uses the gpd_id to identify the group in subsequent calls. Groups are destroyed by calling `mmp_gpd_destroy(gpd_id)`.

A real protection domain can *add* (or join) a group protection domain by calling `mmp_gpd_add(gpd_id)`. If the supervisor allows the domain to add the group, permissions to the group's memory regions are added (or ORed into) the calling domain. This process is shown in Figure 6-2, where PD 1 exports two memory regions read-only to a group protection domain that is added by PD 3. A protection domain can quit a group domain by calling `mmp_gpd_unadd`, making all of the group's memory locations inaccessible to the calling domain.

A group's collection of memory locations grows when a domain exports memory to the group using `mmp_gpd_export`, and it shrinks when a domain revokes a memory region using `mmp_gpd_unexport`. When a domain exports or unexports to a group, the memory supervisor adds or revokes permissions on the new memory for every member of the group. The supervisor sanity checks these calls, making sure that they refer to a group domain which has been created, and that memory unexported from a group was previously exported to the group.

Global permissions (i.e., permissions for all protection domains) are provided by a group

with unrestricted membership. The supervisor can use its knowledge of the permission table layout to share underlying permissions tables if the sharing region meets certain size and alignment restrictions (as discussed in Section 3.4.2). For instance, if a third level table holds information about a 4 KB block of memory that has no restrictions on sharing, mid-level trie tables in other domains can simply point to the owning domain's protection table. The hardware or software that refills the PLB inserts the entry with a PD-ID tag of the currently running domain, not the domain which owns the permissions table from which the entry is loaded. A different table structure (like the one described in Section 3.4.3) would provide support for more efficient global permissions.

Group protection domains were useful in Mondrix and straightforward to implement, even though they were not intended when the hardware was designed. Group domains, like any access control mechanism with groups [SS75], 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.

6.5 Policy for memory ownership and permissions

The memory supervisor policy for memory ownership and permissions is given in Table 6.1. The function names in Table 6.1 do not correspond exactly to the function names in the appendix. The `mmp_` prefix has been dropped to shorten the name so it would fit into the table. The `mmp_mprot` function was split into two logical parts, depending on whether it set permissions for the calling domain (`mprot` in the table), or some other domain (`mprot_export` in the table). The supervisor determines the target protection domain of an `mmp_alloc` or `mmp_free` by reading the cross-domain call stack (as explained in Section 6.6). The arguments to `mmp_pd_subdivide` are simplified.

While there are many details in the table, the supervisor policy follows a few general rules. Export permissions are not explicitly manged, but are folded into ownership and access. An owner can export permissions freely, while a non-owner can export only up to its access permissions level, i.e., a domain's access permissions are its export permissions if it does not own the memory. A non-owner can never dictate permissions to an owner (this policy and its reverse are consistent with the rest of the policy). A non-owning domain cannot downgrade the permissions of another domain. This rule also means that a domain cannot call `mmp_mem_free` on memory it does not own because that call downgrades permissions of the target domain. Because of this restriction, allocator domains must own the memory they allocate, or they must retain access permission in order to retain export permission. If an allocator domain does not own the memory pool that it allocates, then it cannot drop its access permissions on memory it allocates to another domain. The principle of least privilege would dictate that the allocator domain drop its access privileges on allocated memory, since the allocator does not need to access that memory until the domain using it frees it. The lack of explicit export permissions makes the MMP system simpler to implement, but it prevents this application of the principle of least privilege. Finally, a domain must own the memory it uses to create a new domain.

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

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				n	A≤B ? A : ERROR			A non-owner can only downgrade his/her permissions.
y	X	n	X	mprot_export(ptr, len, C, target);	y	X	n	C	An owner can override a domain's permissions.
n	X	y	X		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 his/her 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				n	ERROR			A domain cannot subdivide with memory it does not own.
y/n	X	y	X	pd_free(target);	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. (see Section 6.4.2).
y	X	n	none	mem_alloc(ptr, len);	y	X	n	RW	When the allocator owns memory, it allocates it with read-write permissions.
n	X	y	X		n	ERROR			A domain cannot allocate memory to the memory's owner.
n	F	n	G		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	mem_free(ptr, len);	y	X	n	none	A free revokes permission from all sharing domains.
n	X				n	ERROR			A non-owning domain cannot free memory.

Table 6.1: 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' 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.

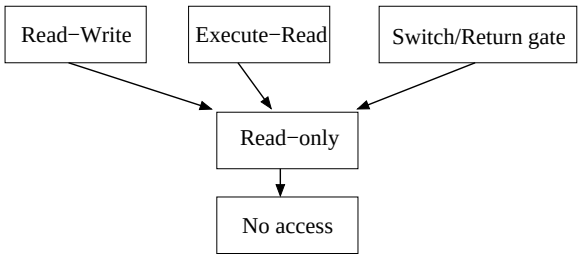


Figure 6-3: A partial order on permissions values.

Table 6.1 refers to an ordering on permissions values. Figure 6-3 shows the partial order that Mondrix uses. Read-write, execute-read and gate permissions all compare as equal, so a non-owning domain can convert among these permissions values (this is the meaning of the line in Table 6.1 for `mprot` with a non-owning caller).

6.6 Dynamic memory allocation

This section describes how the memory supervisor interacts with the system's dynamic memory allocators. The supervisor manages permissions, and the allocator manages memory regions, and these functions should be kept separate. The challenge is to reduce the amount of work the memory supervisor must do during memory allocation and deallocation for efficiency, but give the supervisor enough information and control to set policy for memory use.

A key division of labor between the supervisor and the allocator comes from who tracks the length of allocated memory. The allocators track the length, so the user does not need to provide it when memory is freed. The supervisor should not duplicate the length information which is tracked by the allocator.

6.6.1 Design of a generic memory allocator

Figure 6-4 shows one design for a generic memory allocation service. In the before picture on the left of the legend, there are two domains: a client domain, which owns its code and data, and an allocator domain that has only has code (in practice it would have some static data). The allocator domain owns the memory pool it manages (labeled, "pool" in the figure), but it cannot access the memory in the pool. The state of the system after a successful memory request is shown on the right of the legend. The client has been given read-write permissions by the allocator PD on a block of memory taken from the allocator's pool.

Table 6.1 specifies that a domain which calls `mmp_mem_alloc` on memory it owns, exports read-write permissions on that memory to its caller. The supervisor implements this policy so that it need check only ownership to allow allocation or deallocation to proceed. Memory allocation and deallocation happens frequently. Checking ownership is computationally cheaper than checking export permissions by reading the permissions table (it also requires fewer memory references). Having an allocator own its memory pool enables efficient dynamic memory allocation with the expected semantics that a domain which allocates memory also gains read-write permission on that memory. As a consequence of this

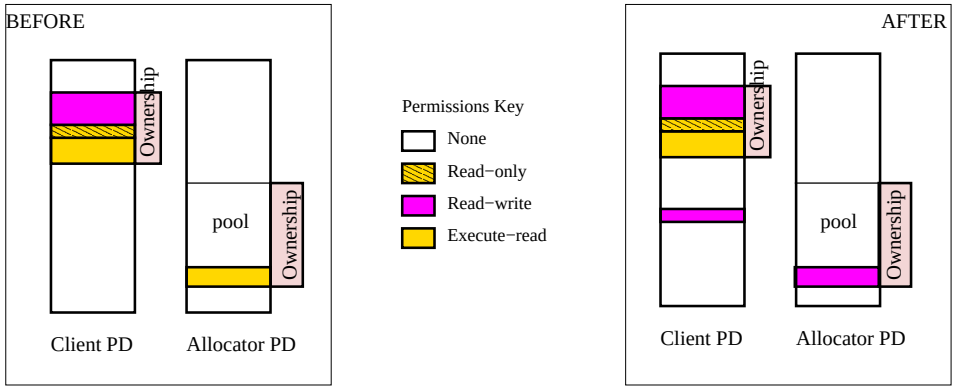


Figure 6-4: A before and after picture for memory allocation. In this example, the allocating domain maintains ownership of allocated memory, but no access permissions. Protection and ownership information is shown. The small spot of read-write permissions in the allocator domain after the allocation is for the allocator to record meta-data into the memory that directly precedes the allocated region

design, the allocator owns all dynamically allocated memory.

6.6.2 Freeing memory

One policy that supports the standard memory allocation model is that all permissions on a region of memory is revoked for all domains when that memory is freed. The memory supervisor revokes the permissions, because it cannot trust the domain freeing the memory to do so. If the program has a bug and fails to revoke permissions on a memory block that it frees, then when the memory is reused, it could access the new block (so could anyone who shared the original), creating a protection breach. In addition, it requires too much additional code to retrofit a legacy application to track what memory it has exported, to whom, and its length information. The supervisor, or the supervisor and the allocator should track this information

The supervisor could simply revoke permissions on memory that is freed from every domain, but that would require writing every domain's protection table. Instead, the supervisor maintains a data structure to track permissions sharing across domains, so it must write the permissions table only of domains that have permissions on the memory region being freed. Section 6.7.1 has a description of the data structure the supervisor uses to track inter-domain sharing.

6.6.3 Dynamically allocated memory and domain deletion

When a program deletes a protection domain, the supervisor revokes permission to access memory owned by the domain from all domains. If another domain has a pointer to memory owned by the deleted domain, accessing that memory will cause a fault. This problem of dangling pointers is common to system which are decomposed into independent services [vECC+99], and its resolution is beyond the scope of this thesis.

Memory dynamically allocated by a domain must be freed when the memory supervisor deletes a domain, to prevent a resource leak. The memory supervisor does not track the

allocating domain for dynamically allocated memory, so it trusts the kernel to free the memory allocated by a domain before deleting the domain.

Nooks [SBL03] uses its fault recovery mechanism to track kernel objects that must be reclaimed when a domain is deleted pragmatically or because it experiences a fault. A fault recovery mechanism for Mondrix should be designed to allow Mondrix to reclaim dynamically allocated memory.

6.7 Memory supervisor data structures

This section describes two important data structures used by the memory supervisor. One tracks the memory sharing patterns of protection domains, the other tracks group protection domains.

6.7.1 Tracking memory sharing across domains

The supervisor tracks which domains are sharing memory, where sharing means multiple domains can access the same memory region. When a program frees a region of memory, the supervisor can revoke permissions on the region from all, and only, the protection domains that are sharing it. Programs free memory frequently, and adjusting permissions tables requires memory accesses and computation, so it is more efficient for the supervisor to track which domains are sharing memory, and only adjust their tables when memory is freed.

The data structure which tracks sharing balances performance with accuracy. Most memory is not shared, but the sharing list must be consulted on every deallocation, and on every call to `mmp_mprot` which exports permissions. The supervisor maintains a short list of protection domain identifiers for each page of kernel memory, indicating which protection domains can access any memory on the page. If domains A and B could both access the word at address `0xC0129874`, the supervisor will revoke permissions from both domains for all memory freed in the range `0xC0129000-0xC0129FFF`. If the sharing list grows too long, the page is considered global and permissions are revoked from every domain whenever memory is freed anywhere in the page.

A simpler scheme could have the supervisor maintain a list of all domains to which each domain has exported permissions. The supervisor would track if domain A has exported memory to domains B and C. The supervisor would then revoke access rights from B and C whenever domain A freed memory. Early on, Mondrix used this scheme, but it is quite inefficient since the kernel exports memory to every other domain and also allocates and frees the most memory.

6.7.2 Tracking group protection domains

Group domains can contain many disjoint memory regions, and the memory regions that constitute the group can change frequently. An SST (see Section 3.1) is not an efficient data structure for permissions regions that change frequently, because on average half of the records need to be recopied on every insertion or deletion to retain physical contiguity.

Group protection domains use a data structure which maintains an SST for every page in the kernel virtual address space. This ensures that no SST gets too long, so insertions

and deletions are efficient. Each page's SST can represent the entire region that passes through that page. For example, the SST for page `0xC123A000` can have an entry which extends past `0xC123B000`. To determine if an address is in a group, the supervisor uses the page number of the virtual address to index the array of SSTs, and then searches the SST for that page. Updates write the SST of each page which is intersected by the region being written.

One subtlety that arises with using this data structure is that information about positive permissions that extend beyond the page that owns a particular SST can be trusted, while information about a lack of permissions beyond the page that owns an SST cannot be trusted. Every SST ends with an entry that specifies no access permission until the end of memory. This entry should only extend to the end of the page. SSTs from succeeding pages must be consulted individually to find the next regions with permissions.

The group protection domain implementation restricts any memory location to belong to at most one group. This restriction simplifies the algorithms for group maintenance, allowing efficient determination if a given memory range intersects any memory managed by a group.

Chapter 7

Mondrix: the MMP-Enabled Linux Prototype

Operating systems written in unsafe languages are efficient, but they crash too often. OS crashes are worse than user software crashes because an OS crash requires a time consuming reboot and may cause multiple users to lose data. With important, non-transient data being stored on laptops, desktops and a proliferation of embedded devices (like PDAs and digital cameras), lack of reliability translates directly into personal inconvenience. Crashes and security breaches incur large costs in lost productivity, increased system administration overhead, and large business cost. We believe system reliability should be a bigger goal for OS developers, and we believe that computer architects can do more to help OS developers write robust software. Page-based protection is not adequate because of high context switch penalties and inefficient use of memory when modules have complex sharing patterns. Better memory protection is needed today.

We use Linux as a sample application for MMP because Linux supports software modules for extensibility (see Section 1). MMP protection domains can isolate these modules, making failures easier to detect and to recover from. Preventing failures in the operating system increases system reliability. Linux is a huge and mature code base, so adapting it to use MMP provides evidence that MMP's abstractions are useful for software, even software that was not developed with MMP in mind.

The main challenges to adapt Linux to use MMP are modifying the module loading process to load modules into independent protection domains; modifying the memory allocation process to be domain aware; and modifying the kernel and its modules to explicitly manage memory access permissions and cross-domain calls. This chapter describes our approach to these challenges.

Module maintainers would probably do a better job of demarcating permissions (and possibly rearranging data structures to make such demarcations more efficient) than I did. However, the prototype demonstrates that a non-expert can add permissions information relatively quickly based on the usage patterns present in the kernel.

We call our Linux prototype, "Mondrix." Features and algorithms that are common to Linux and Mondrix will be ascribed to Linux, while Mondrix will denote code that is new or modified for MMP support.

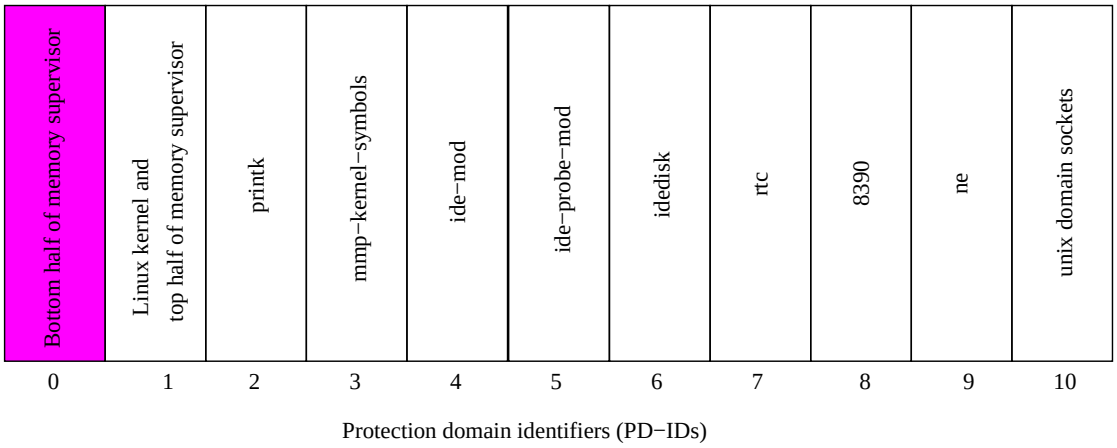


Figure 7-1: How Mondrix loads different modules into different protection domains. The bottom half of the MMP memory supervisor is a layer of software that manages the MMP hardware, and has full access to all of memory, and so resides in PD 0. The top half of the supervisor resides in PD 1, with the bulk of the Linux kernel. `Printk` is a kernel subsystem loaded into PD 2. Kernel modules reside in subsequent protection domains, and all modules loaded into Mondrix are shown.

Figure 7-1 shows how Mondrix uses protection domains. It is similar to Figure 6-1, but the kernel and top half of the memory supervisor share a protection domain. The supervisor and the kernel should not share a protection domain, but time for implementation ran out before the top half of the supervisor could be separated from the rest of the kernel.

7.1 From system reset to kernel initialization

At system reset, the processor starts running at the reset vector in domain 0. The BIOS loads the bottom half of the memory supervisor into physical memory and transfers control to it, letting it know how much physical memory is in the machine. Early on, the supervisor establishes a handler for hardware permission faults.

Once initialized, the supervisor creates a new domain (PD-ID=1) to hold code and data for the core of the kernel. The supervisor transfers ownership of most of physical memory to the kernel, retaining enough memory to manage the permissions tables. A more general approach would have the supervisor partition physical memory among different OSes or virtual machines or other pieces of code, based on some a priori specification.

To start the kernel, the supervisor first loads the boot loader into PD 1. It initializes cross-domain calling by allocating memory for the cross-domain call stack, and initializing the `CDST` (cross-domain stack top) register. Then it does a cross-domain call to the boot loader entry point in PD 1. The boot loader loads the kernel image into physical memory, and returns to the supervisor. The supervisor reads the kernel symbol table to set proper permissions for the kernel text section, read-only data, etc. The supervisor then starts the kernel by doing a cross-domain call to its entry point.

7.2 Loading modules into protection domains

One of the primary uses of MMP is to enforce modular boundaries that are already present in software systems. MMP isolates modules by allowing each module to reside in its own protection domain. Linux kernel modules are object files that a user loads into a running kernel. Mondrix loads each module into its own domain, as shown in Figure 7-1.

To load a module into the kernel address space, a user calls the `insmod` program. `Insmod` acts like a dynamic linker, resolving the undefined symbols in the object module against the currently running kernel. For example, if a module has `vmalloc` as an undefined symbol, `insmod` resolves that symbol to the physical address of the first instruction of the kernel's implementation of `vmalloc`. `Insmod` prepares a module in user space, then passes the image of the prepared module (whose initial bytes look exactly like a `struct module`) to the kernel module loader via a system call. The kernel module loader sanity checks the module, (e.g., making sure it isn't bigger than the available memory, that pointers in the `struct module` structure do not point outside the boundary of the module, that the address of the module's initialization routine (held in the `struct module`) is in the module's code segment, and other consistency checks) and then calls the module initialization routine.

7.2.1 Modifying `insmod`

The MMP memory supervisor needs two pieces of information, that `insmod` provides, to initialize a kernel module. It needs to know the length of the program sections in the module, and it needs to know the start and end location of every function in the module. It needs program section information to properly set the initial permissions for the module (e.g., execute-read for text). It needs function length information for all public functions (functions that can be called from outside the module) so it can guarantee that switch and return gates are set only at the start and end of a function, respectively.

Linux's `insmod` inserts symbols into the prepared module that indicate the boundaries of the program sections, and the MMP supervisor reads those symbols.

The supervisor also needs to know the start and end of every public function in the module that `insmod` is loading. Linux's `insmod` puts symbols in the prepared module indicating the start of every publicly exported function, but it does not include length information. Mondrix's `insmod` reports the location of every function symbol in a module, public and private, which allows the supervisor to determine the length of all public functions. Figure 7-2 shows the layout of two kernel functions. In this case, the public function `kmem_cache_destroy` is followed by the private function `kmem_cache_grow`. The supervisor determines the length of a function by finding the distance to the next symbol.

The start of the succeeding function does not always indicate the end of the previous one. The kernel compilation process places the first instruction of a function on a word aligned address for performance reasons. It inserts `nop` instructions between the end of one function, and the word-aligned beginning of the next, as seen in Figure 7-2. The supervisor searches backwards from the succeeding symbol (`kmem_cache_grow` in the example) for the return instruction by disassembling the code, and recognizing compiler idioms.

Mondrix could have left `insmod` unmodified, and used its symbol information to find all public functions, and then disassembled the functions until finding their exit point. While separating code from data is not a computable problem in general [Cif94], most modern compilers separate code and data sufficiently to allow disassembly. Given the complexity of disassemblers for the x86 instruction set, we rejected this option, and instead augmented the

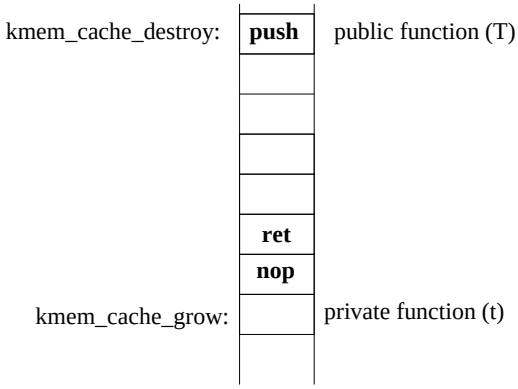


Figure 7-2: The layout of two kernel functions is shown. `kmem_cache_destroy` is a public function (`nm` uses the symbol T for such symbols), and it is followed by `kmem_cache_grow` which is a private function (t in `nm` notation). There is a padding `nop` between the last instruction of `kmem_cache_destroy` and the start of `kmem_cache_grow`.

symbol information provided by `insmod`. The limited disassembly that the supervisor must do to skip `nops` and find the return instruction is vastly simpler than a full x86 disassembler.

7.2.2 Domain creation with module loading

The Mondrix kernel gets ownership of physical memory from the BIOS during the boot process. It then subdivides itself (using `mmp_pd_subdivide`), to isolate major subsystems in their own domains. While `pd_subdivide` is general enough to be the only domain creation function, a convenience interface to load kernel modules into a new domain is also provided.

A kernel module is an object file that contains standard Unix program sections, like text, read-only data, read-write data and read-write bss. `mmp_module_init` takes a parameter of type `struct module*`, the data structure the kernel uses to track modules. This routine subdivides the calling domain, loading the kernel module in the new domain, and setting the permissions on the module to the values specified in the object file (e.g., execute-read for the text section). It combines domain creation with module loading.

Figure 7-3 shows permissions and ownership information for domain creation with module loading. 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.

7.2.3 The `mmp-kernel-symbols` module

The MMP memory supervisor gets function length information from modules when they are loaded via `insmod`, but what about functions in the kernel itself? The kernel's access to its own symbol table is limited. Symbols to demarcate program sections can be put in the loader map (`vmlinux.lds`), and can then be referred to with extern declarations. But there is no convenient way to use this mechanism to put the address of every function into the kernel in a way that could be read as a data structure.

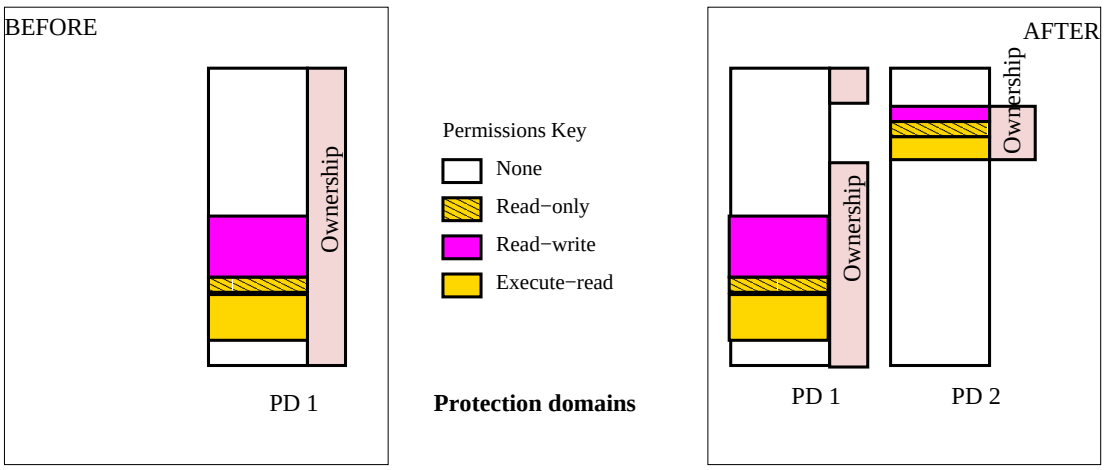


Figure 7-3: A before and after picture for domain creation with module loading. For each domain, the thicker bar shows the protection information, and the thinner side bar shows ownership information.

Mondrix gets kernel symbol information the same way system utilities like `ps` and `top` get it—via the `System.map` file, commonly found in the `/boot` directory on Linux systems. The kernel build process generates this file by running `nm` on the kernel image.

The memory supervisor cannot read the `System.map` file directly because Linux kernel code cannot read files, it needs a process context for file access. The Linux kernel always needs some user-level program to read files; Mondrix uses `insmod` because it already needed a modified `insmod`. Mondrix modifies `insmod` to recognize a special module (called `mmp-kernel-symbols`). When it prepares that module for insertion into the kernel, it first reads the `System.map` file and inserts all of the symbol information from that file into the `mmp-kernel-symbols` module.

The symbol information from the `System.map` file contains the locations of all kernel functions, allowing the memory supervisor to check that gate permissions are only set on the first or last instruction of kernel functions.

The `mmp-kernel-symbols` module is loaded before any other module, so the memory supervisor can check the gate permissions for kernel functions imported by successively loaded modules.

7.2.4 Setting permissions on kernel program sections

The kernel and its modules, like user-level executable programs, have several program sections. From a protection perspective, the important sections are the code (text section), the read-only data section, and the combination of the read-write data, and read-write bss sections.

In general, the memory supervisor places the obvious permissions on each section (e.g., execute-read permission on the text section). However, there are a few interesting complications. The first are the interrupt request (IRQ) handling vectors in the kernel itself. These are pieces of code that are jumped to by the processor when it receives an interrupt from an external device. They are executable code but they reside in the read-only data section. The memory supervisor must mark them as code, not as read-only data.

Code in the `.text.init` section is read as well as executed. The data for an initial page table is also read from the executable section. This code can be marked execute-read, but not execute-only (MMP does not support execute-only, so it uses execute-read).

In one case, the module `ide-disk` has many static strings, which are read by the module `ide-mod`. Instead of finding and exporting every string, the module exports its entire read-only data section. The memory supervisor knows about section boundaries based on symbols placed in the module by `insmod`. The memory supervisor exports the `mmp_get_my_ro_section` method which allows the `ide-disk` domain to export its entire read-only data section to the `ide-mod` domain.

7.2.5 Communicating memory sharing patterns to MMP

Once the kernel loads a module, the module must define its relationship to the rest of the system. It needs to ask permission to call certain services, it needs to export permission on the services it provides, and it might need to import or export permissions on data relating to those services. The MMP system must be notified about these intended sharing patterns. Because MMP is intended for legacy systems, having the MMP system figure out the sharing patterns without requiring programmer effort is desirable. We present an approach which uses symbol information to infer sharing patterns, and explain how it works better for code than data.

```
slab.c                                ide.c
  void* kmalloc(size_t, int) {...       extern void* kmalloc(size_t, int);
ksyms.c
  EXPORT_SYMBOL(kmalloc);

timer.c                               sched.h
  unsigned long volatile jiffies;     extern unsigned long
ksyms.c                                                 volatile jiffies;
  EXPORT_SYMBOL(jiffies);

rtc.c                                 file_table.c
  static int rtc_release (              if(file->f_op
              struct inode* inode          && file->f_op->release) {
              struct file* file) {...     file->f_op->release(inode, file);
  struct file_operations rtc_fops = {
    release: rtc_release, };

ide-probe.c                           ide-disk.c
  drive->id = kmalloc(                  idedisk_setup(ide_drive_t *drive) {
               SECTOR_WORDS*4,            struct hd_driveid *id = drive->id;
               GFP_ATOMIC);               if(id->max_multsect) {
```

Figure 7-4: Named and anonymous sharing of code and data. The source file and defining code snippet appears on the left. The source file and corresponding using code snippet appears on the right.

Figure 7-4 shows examples of how code and data can be shared by name, or anonymously. The snippet on the left shows the definition, and the snippet on the right shows the corresponding use. The first example shows the routine `kmalloc` is imported via an `extern` declaration. The second example shows the data item `jiffies` is imported by an `extern` declaration.

In C, symbols (code or data) that are not declared static may be imported by modules that are linked into the same address space (via the `extern` statement). The kernel programming environment offers a way to limit the code and data that is visible to a loadable kernel module. Symbols exported to modules (and symbols exported from the module to the kernel) must appear in an `EXPORT_SYMBOL` directive. This directive puts the symbol name and location in a special symbol section of the binary. `Insmod` looks in this section to do its symbol resolution.

The second two examples in Figure 7-4 show anonymous export for both code and data. For code, a pointer to a function that is declared `static` is placed in a structure, and passed to the kernel, which calls the function via the pointer. The name of the function is not involved.

Anonymous export of data is common. In the example, a piece of data is dynamically allocated and assigned to a pointer that resides within a structure. A pointer to that structure is passed between modules, and the other module finds the data by traversing pointers.

There is programmer effort required to notify the MMP memory supervisor how memory is shared between different modules and between a given module and the kernel. We hoped to reduce this programmer effort by using symbol information when code or data is shared by name, but our results with this approach are mixed. For example, while the supervisor places gate permissions on functions exported by name, it relies on the programmer to explicitly call `mmp_func_gate` for functions exported by an anonymous pointer. While exporting symbols by name is the currently encouraged method for symbol export in the Linux kernel, there is still plenty of legacy code which export functions using anonymous function pointers.

Named export for data is more rare than for code, so using symbol information to automatically resolve data exports does not work well. Most data export needs to be explicitly managed by the programmer. Symbol use does capture some data sharing patterns, but then it can be confusing for the programmer to see most data explicitly exported by module code, while other data is omitted and exported implicitly by the MMP supervisor. The programmer must remember that the omitted data was exported implicitly via the `EXPORT_SYMBOL` mechanism rather than explicit calls to the memory supervisor.

The largest disadvantage of using data symbols to indicate memory sharing relationships is that importing by name is likely to be too generous. For instance, the `ide-probe` module imports the symbol `ide_hwifs`. This symbol is the master array containing all of the information for all EIDE devices attached to the machine. In fact, `ide-probe` needs only a subset of this information.

A related disadvantage to this problem is C's ambiguity between pointer and array. Often a single record, or a small number of records in an array must be exported, and it is overly permissive to export the entire array. However the symbol does not provide enough information to appropriately limit the permissions.

7.2.6 Initial RAM disk

Mondrix make Linux more robust by isolating modules in their own protection domain, so Mondrix wants to load the EIDE disk driver as a module. But the root disk is an EIDE disk, so EIDE disk support is necessary for a successful boot. The RAM disk support in Linux lets Mondrix resolve this catch-22. RAM disk support allows the kernel to treat a block of memory as a disk device. By placing the EIDE driver on an initial RAM disk (and compiling RAM disk support into Mondrix), the OS startup script (`linuxrc`) loads the EIDE modules, and then pivots the root file system to the real disk to continue Mondrix initialization.

7.2.7 The `printk` domain

`printk` and its related functions, (e.g., `sprintf`, `snprintf`, `vsnprintf`, etc.), have enough special memory access requirements to merit their own domain. These functions are unique because they must read string arguments from whoever calls them. Additionally, `sprintf` and `vsprintf` must write into buffers whose length is not specified.

Explicit export of every string to the printk domain would be tedious, and not beneficial for the robustness of the kernel. Mondrix gives the printk domain permission to read most of kernel memory (excepting kernel code). This approach compromises isolation because the printk domain can read all kernel data, but the functions within the printk domain are small and easily audited, and most of the kernel assumes they can read (and write) their string data already. The MMP protection tables support large permissions regions efficiently, so giving the printk domain read-only access to a large address range does not compromise the performance of Mondrix. (The printk domain would not need read permission on so much of the kernel address space, if the kernel compiler put all all static strings in a special program section. In this case, the printk domain would need read access only to this section in the kernel and its modules).

Mondrix does not fully address safety for the `sprintf` family of functions, for instance, the prototype lets every domain read and write the kernel stack (details in Section 7.4.3). However, it implements one technique which could be part of a more complete solution.

By special arrangement with the supervisor, the functions in the printk domain are allowed to get read-write permissions for a memory block whose length is determined by reading the permissions table of the domain that called the printk domain function. If the disk driver calls `sprintf` with a buffer whose address is `0xC012A000`, `sprintf` calls the supervisor (via the `mmp_printk_rw` function) which reads the disk driver's permissions table. The supervisor might find that the disk driver has read-write permission on the (page-sized) range `0xC012A000--0xC012AFFF`, but that it has no access permission on `0xC012B000`. The supervisor will then grant `sprintf` read-write permission on the memory block from `0xC012A000` to `0xC012AFFF`. Before `sprintf` returns, it calls `mmp_printk_done`, which returns the printk domain's access permissions to whatever they were before the call to `mmp_printk_rw`.

Limiting the permissions for `sprintf` in this way minimizes, but does not eliminate, the possibility of data structure corruption. If the calling domain had access to two pages in a row, this technique would give `sprintf` permission to write both pages, even if the pages held distinct data structures.

The Linux implementation of `sprintf` does write a byte at a time, so a single inaccessible word between data structures would be sufficient to prevent buffer overrun. Using our technique, if the allocator insured the presence of an inaccessible word between each

allocation (as was done for `malloc` in Section 5.4), then this technique would guard again buffer overruns in `sprintf`.

7.3 Dynamic memory allocation in Mondrix

The last chapter (Section 6.6) described how a generic memory allocation service would interact with the MMP memory supervisor. This section describes the details of how Mondrix integrates the MMP memory supervisor with the dynamic memory allocators in Linux.

7.3.1 Background on Linux's memory allocators

There are two major dynamic memory allocators in Linux, and one minor allocator, used only during the boot process. The first major allocator is the slab allocator [Bon94], which is accessed via `kmalloc` and `kfree` as well as `kmem_cache_alloc` and `kmem_cache_free`. This allocator is used for small to mid-sized objects, up to 4 KB. It manages entire pages (slabs), and splits the page into equal, power-of-two sized regions. It keeps its accounting information in an unused area on the page.

The second main allocator is the buddy allocator, which is accessed via `vmalloc` and `vfree`, as well as `alloc_pages`, `__get_free_pages` and `free_pages`. This allocator is used for larger objects, greater than or equal to 4 KB.

Linux uses the minor allocator, located in `bootmem.c` only during the boot process to allocate and free memory. During `mem_init` any remaining memory is freed en masse, and put on the buddy allocator's free list. The memory use of this allocator can either be ignored, or tracked by the memory supervisor.

Mondrix tracks the allocations made by the bootmem allocator, which creates a chicken and egg problem since the memory supervisor is a client of the slab allocator which has not been initialized when the bootmem allocator is active. To resolve the issue, the OS initializes the memory supervisor in two phases, via calls to `mmp_early_init` and `mmp_init`. The OS calls `mmp_early_init` early in `start_kernel` (the Linux kernel startup function), before the bootmem allocator is used, and before the slab allocator is initialized. The kernel calls `mmp_init` after the slab allocator is operational.

This chapter will refer to these three allocators as Linux's memory allocators. In Mondrix, they are all resident in the kernel's protection domain, though they could reside in their own domain (which would incur only a small additional cost as cross-domain calls which do not change the PD-ID become cross-domain calls which do change the PD-ID).

7.3.2 Integrating the memory supervisor with Linux's memory allocators

There are four challenges to integrating Linux's kernel memory allocators with the MMP memory supervisor: the allocators must update their data structures and call the supervisor atomically, the allocators must provide length information to the supervisor when memory is allocated and when it is freed, the supervisor should not do much computation during allocation or deallocation, and the supervisor should support kernel allocators for custom data structures.

7.3.3 Executing the allocator and memory supervisor atomically

On uniprocessors, Linux's allocators disable interrupts so they execute atomically. Mondrix modifies the allocators to call into the supervisor (via `mmp_mem_alloc` and `mmp_mem_free`) once the allocator has updated its internal data structures, but before interrupts are re-enabled.

```
Allocator decides that memory region X is free.
  Interrupt
  Memory region X is reallocated
  End interrupt
Allocator calls mmp_mem_free on memory region X.
```

Figure 7-5: Memory permissions corruption scenario if dynamic memory allocator and memory supervisor do not execute atomically.

If the allocator does not call the memory supervisor with interrupts disabled, the resultant race condition will cause the supervisor to set the wrong permissions on memory. Figure 7-5 shows a sequence of events that cause permissions corruption if `mmp_mem_free`) were called after the allocator re-enabled interrupts. The kernel calls the allocator to free some memory (call it memory region X). After the allocator updates its data structures, it re-enables interrupts, and calls `mmp_mem_free`, but before `mmp_mem_free` can disable interrupts, the system takes an interrupt. Memory region X is reallocated during the servicing of the interrupt. When execution of the original thread resumes, `mmp_mem_free` would incorrectly revoke permissions on memory region X—that memory is now in use. The author observed this race condition in an early implementation of Mondrix. Even if `mmp_mem_free` itself disabled interrupts, the corruption in Figure 7-5 could happen. On multi-processor systems, the allocator must call the supervisor while the allocator spin lock is held.

7.3.4 Providing length information to the memory supervisor

Requiring the memory allocator to pass the length of the memory being allocated and freed means the supervisor does not need to duplicate the (considerable) work of tracking allocation lengths. Determining the length of an allocation is straightforward because the user provides the length. Determining the length of a memory region being freed depends on the details of the allocator. The slab allocator is interesting because it rounds up the size of a memory request to the nearest power of two. If the client requests 10 bytes, the allocator provides 16 bytes. However, the allocator passes a length of 10 bytes to `mmp_mem_alloc`, and a length of 16 bytes to `mmp_mem_free`. There is no problem if the length being freed is larger than the length being allocated. By only granting permissions on a subset of a memory region (e.g., 10 of 16 bytes), the allocator provides a "red-zone" of inaccessible memory before the next memory block (which starts at a 16 byte offset).

7.3.5 Reducing memory supervisor work during (de)allocation

Memory allocation and deallocation happens frequently, and so it must be efficient. Linux's memory allocators are highly optimized. Mondrix adds work to both allocation and deallocation because the memory supervisor must set or revoke permissions. Mondrix uses the

design shown in Section 6.6.1, where the allocator owns its allocation pool. This reduces the work the memory supervisor performs in the common case to determining ownership for the caller of `mmp_mem_alloc` or `mmp_mem_free`. Then the supervisor writes the permissions table of the domain which called the memory allocator.

If the memory supervisor is told the location of the dynamic memory pool, the ownership check is reduced to a simple range check. Even if many domains are created, dynamic memory allocation can still proceed quickly because the ownership test is independent from the number of protection domains. Mondrix implements this optimization.

7.3.6 Supporting custom allocators

The memory supervisor interface is general enough that allocators for custom kernel data structures can also use it (e.g., the `sk_buff` allocator in the Linux network code). An `sk_buff` is a data structure that holds packet data. This data structure has a custom allocator in `sock.c`, which includes a private free list. A client domain can call the custom `sk_buff` allocator, which calls `mmp_mem_alloc` to give permission on the `sk_buff` to its caller. Occasionally, the `sk_buff` allocator must call one of the standard Linux allocators to get more memory to carve up into `sk_buffs`.

As mentioned above, the Linux allocators own dynamically allocated memory, so the `sk_buff` allocator does not own the memory it allocates. Table 6.1 shows that the memory supervisor uses a domain's access permissions to determine its export permissions on memory it does not own. Therefore, custom memory allocators cannot drop their access permissions on memory they allocate if they expect to reallocate the memory.

When a custom allocator calls `mmp_mem_alloc` or `mmp_mem_free`, the memory supervisor will determine that the domain does not own the memory so it will read the protection table of the custom allocator's domain. This is less efficient than the main allocators which use the ownership test, but custom allocators are not used as much as the general memory allocators.

Finally, Linux's allocators and all of the kernel custom allocators reside in protection domain 1 in Mondrix; however, this fact was not used to optimize the custom allocators.

7.3.7 Trusting the caller of `mmp_mem_free`

The current Mondrix implementation trusts that the domain that frees a region of memory is the domain that allocated it. Mondrix could check the kernel's behavior by tracking the domain which performs the allocation and verifying that the same domain performs the free. The allocator is the easiest place to track the domain, because it already tracks the length of allocated memory regions. Giving the allocator the responsibility to record and verify the allocating domain makes the allocator part of the MMP trusted computing base.

7.4 Managing permissions in Mondrix

Mondrix tries to give kernel modules the minimum amount of memory permissions that still allow the module to function properly. The two most important groups of Linux modules that Mondrix modifies is the EIDE disk driver, and the (NE2000) network driver. This section describes how these subsystems share memory in Mondrix, and how Mondrix

balances giving modules only the minimal amount of permissions they need with other factors like ease of programming and performance.

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`). The NE2000 network driver has a top half (`ne`) to manage the reception and transmission of packets, and a bottom half (`8390`) which manages the details of moving data onto and off of the network card, and handles device interrupts and device initialization. Different halves of a driver share data structures, and call each other frequently.

The different modules in a driver must communicate their sharing patterns to the memory supervisor by exporting permissions to each other. Sometimes finding the minimal subset of memory that two modules must share is difficult. For example, in the `ide-mod` module, finding the subset of the `ide_hwifs` array that the `ide-probe-mod` module absolutely needs is a tedious and error prone process.

Determining the sharing pattern among modules in a driver is difficult, but simply loading the modules into different protection domains, and then running the system until a permissions violation occurs is an effective way to make progress in determining the proper sharing relations. While this is an ad hoc technique, there is no general method to guarantee that the sharing patterns described to the supervisor are the minimal (or even sufficient) set necessary for a module's function. Perhaps model checking or a specification language would allow more formal verification.

7.4.1 EIDE disk driver

The most difficult part of adapting the disk driver is determining when write permission can be revoked from the driver. The EIDE driver writes pages in the page and buffer cache. Finding the exact program points where the disk driver need to gain write permission to those pages, and then where they should drop those permissions is challenging, so Mondrix gives the disk driver permission to write pages in the page or buffer cache when they enter the cache, and revokes permissions when the pages are freed. This policy compromises isolation by allowing pages in the buffer cache to be corrupted by a poorly behaved EIDE disk driver, but all other kernel data structures are protected from the disk driver.

7.4.2 NE2000 network driver

Mondrix controls the permissions on network buffers because the scope of their use is obvious. The kernel grants read permission on the data portion of an `sk_buff` (the kernel data structure which holds packet data) just before the driver transmits a packet, and the kernel revokes the permissions right after the transfer. The kernel grants read-write permission on the data portion of an `sk_buff` just before the driver receives a packet, and the kernel revokes the permissions right after the receive. The NE2000 driver does not use DMA, making the live range obvious. This policy tightly restricts the permissions of the network driver, at the cost of frequent table updates. If the driver did use DMA, the live range of the packet data would extend from when the kernel scheduled an operation to the interrupt that signals that operation's completion.

If the packet buffer memory is not used for any other purpose before it is freed, the

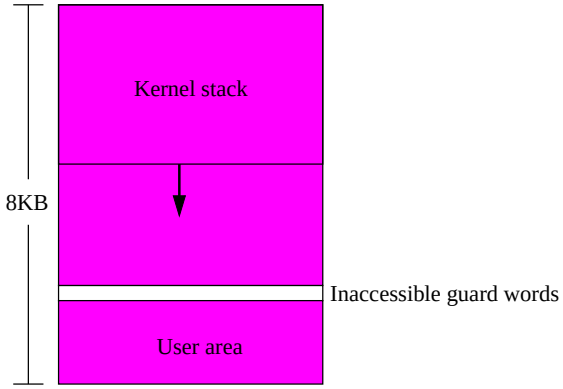


Figure 7-6: How MMP can protect the user area from the kernel stack. The user area and kernel stack occupy the same two page region, with the kernel stack growing down toward the user area. MMP would enable inaccessible guard words between the kernel stack and user area to increase the likelihood of a protection fault if the stack grows into the user area, rather than the silent corruption that Linux kernel developers fear today.

network scheduler could eliminate the permissions revocation after the packet arrival. The network driver domain then accumulates permissions on many regions before the permissions are revoked by a free. A wild write during the execution of the driver might overwrite data from a previous packet, but this level of risk might be acceptable. Lazy permissions revocation eliminates one update to the permissions table, but increases the window of vulnerability. This policy was not implemented in Mondrix, the more restrictive policy was implemented to avoid the window of vulnerability and to stress the MMP system.

7.4.3 Kernel stack/user area

In Mondrix, the memory supervisor places the kernel stack of every process in a global group protection domain, giving every domain read-write permission on every kernel stack. Allowing every domain read-write access to all kernel stacks simplifies Mondrix because domains do not manage permissions on stack allocated memory, it is all read-write. However, it leaves open the possibility of stack corruption, which is a big problem in practice. Managing stack permissions in MMP requires additional mechanism and chapter 9 presents a design.

In Linux, both the kernel stack and the user area both occupy the same two page memory region, with the kernel stack growing down toward the user area, as shown in Figure 7-6. The user area (`struct task_struct` is a structure which contains many system details for a process, like its scheduling state, its memory layout, and its user credentials. Linux kernel developers are encouraged to not increase the size of the user area, because that increases the chance that the kernel stack will grown down into it, silently corrupting it. With MMP, we can add a guard word (or several guard words), which would greatly increase the chance of a memory protection fault should the stack grow down into the user area. This ability to detect corruption also helps justify putting the cross-domain call stack in the user area (as discussed in Section 4.3.2), because the user area is no longer likely to be silently corrupted.

7.4.4 Optimizing PLB performance for kernel stack/user area

A Linux processes' kernel stack is referenced often, as is its user area. Keeping entries for these data structures resident in the PLB will reduce its miss rate, so these entries should not be considered for replacement along with every other entry. We statically partition the PLB into a large area that uses random replacement (60 entries), and a smaller area that uses FIFO replacement (4 entries).

When the kernel schedules a thread, it notifies the memory supervisor, which writes two registers, `wire_base` and `wire_bounds` which are the base and bound registers for the wired region. Any PLB refills from the wired region are put into the FIFO replacement policy part of the PLB by the PLB miss handler. As we guessed in [WCA02], four is a good number of entries for this part of the PLB.

We could use software to wire in the entries from the kernel stack and user area into the smaller part of the PLB (as we suggested in [WCA02]). However, it can be difficult to predict exactly which areas of the user area will be referenced, so our prototype uses the slightly more sophisticated hardware model of the two wired registers.

7.4.5 Optimizing function pointers

One common idiom in the kernel is to call services via a function pointer. Often, memory must be exported to the domain that implements the function. While it is possible to look up the owner of the function pointer to find the implementing domain, it is expensive: using the supervisor function `mmp_code_to_pd`), this lookup requires a $log(N)$ search through the list of owners (where N is the number of regions in the memory supervisor's owner list).

Consider this example from `fs/file_table.c`.

```
inline pd_id_t find_release_pd(void* release_func) {
   if(release_func == rtc_release) {
      return rtc_pd;
   return mmp_code_to_pd(file->f_op->release);
}
...
if (file->f_op && file->f_op->release) {
   pd_id_t release_pd = find_release_pd(file->f_op->release);
   // rtc.c:636 rtc_release reads file->f_flags
   if(release_pd != kern_pd)
      mmp_mprot(&file->f_flags, sizeof(file->f_flags), PROT_READ,
                release_pd);
   file->f_op->release(inode, file);
   if(release_pd != kern_pd)
      mmp_mprot(&file->f_flags, sizeof(file->f_flags), PROT_NONE,
                release_pd);
}
```

The kernel domain exports read permission to the domain that implements the release operation for the `file` variable (which is of type `struct file*`). In Mondrix, the real time clock domain is the only one that uses this processing path. Therefore there is a check in `find_release_pd` to see if the `release_func` function pointer is the real time clock's implementation of `rtc_release`. This check succeeds all of the time in Mondrix, avoiding

the call to the more costly function `mmp_code_to_pd`. However, the more costly function is included in case a new module is loaded which also implements the release function. In this case, the `release_func` function pointer will not point to `rtc_release`, but will point to the new module's implementation. The check in `find_release_pd` will fail, and the program calls `mmp_code_to_pd` to find the PD for the new module. The most efficient alternative for a small number of possibilities is to use the function pointer to lookup its protection domain via a series of tests, or a small hash table.

7.4.6 Runtime adjustment of permissions

One attractive property of MMP is that the system can adjust the runtime penalty of permissions checking during execution by changing the granularity of permissions regions. A developer might want fine-grained memory protection to help diagnose some intermittent problem. Initially, he would like the system to run fast so its code is positioned to where he has observed the problem. Permissions regions can be coarse-grained during positioning, because the developer does not expect any problems.

When the system reaches the point where the developer has seen the problem, he can reconfigure the system (by flushing the PLB) for fine-grained memory protection. For instance, the developer might place inaccessible words between allocated regions (as in Section 5.4) in an attempt to expose the memory corruption which causes the problem he observed. (To stay within the C language programming model, no data can move during the process of making permissions more fine-grained, so the allocators must have already inserted the guard words, and they must know where they are.) Fine-grained memory protection can be useful for problem diagnosis, but it can be applied selectively during system execution.

7.5 Cross-domain calling

This section discusses issues relating to cross-domain calls in the Linux prototype, specifically: interrupts, argument passing, and inlining.

7.5.1 Interrupts

Handling device interrupts is an important task for an operating system, and MMP allows them to proceed in a protected way. Linux on the x86 responds to 16 device interrupts. A table lists, for each interrupt, the address to which the processor will transfer control when it receives that interrupt. The assembly stubs for each of these entry points call a common routine, which sets up the stack for a routine written in C, which actually responds to the interrupt.

Interrupts do not cause a protection domain switch. All of the assembly stubs for the interrupt entry points have executable permission in every domain. The function call from the interrupt stub to the C handler routine has a switch gate. This technique makes the interrupt assembly stubs a shared library, albeit a simple one that has no data.

Because the assembly stubs are resident in every protection domain, a bug in the code could affect any domain. Luckily, the assembly stubs are on the order of tens of instructions with two or three branches. 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.

7.5.2 Passing arguments

A goal of MMP is to support the same model for passing arguments in cross-domain calls that is supported by a standard function call in C. Function calls in C pass arguments by copying data in registers or on the stack, and by passing pointers to heap-allocated or stack-allocated storage. No additional mechanism is needed in MMP to support passing arguments in registers, since data in registers is checked by MMP when it is loaded from or stored to memory.

The supervisor functions that manage memory permissions are intended for use on heap-allocated memory. Heap-allocated data structures do not need to be marshaled for a cross-domain call, domains can set permissions on shared data structures in advance of the cross-domain call. For example, in a producer-consumer relationship, the producer would maintain read-write access on a buffer and flag value, while the consumer has read-only access on the buffer and read-write access on the flag. Once the permissions are established, they do not need to be modified for every call.

Stack storage is different from heap data because stacks are used by threads that move between protection domains. Every domain has read-write permissions to the kernel stacks in Mondrix, so the protection domain structure does not isolate stack-allocated data structures, leaving them vulnerable to corruption. We present a design for extending MMP protection to stack storage in Chapter 9.

7.5.3 Inlined functions and protection domains

In C, header files sometimes include inlined functions that reference a module's internal data. Any domain which 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.

Sometimes inlined functions are un-inlined. For example `mntput` exposes an internal field of a `struct vfsmount*` in its implementation just for performance, so this function is un-inlined to increase modularity. In other cases, extra permissions are exported from the provider to the client in order to support the inlined test. `down` in `asm-i386/semaphore.h` is a hand-tuned piece of assembly, so the kernel, which owns the semaphores, exports write permission on them to modules that call this function.

Chapter 8

Experimental Evaluation of Mondrix

This chapter analyzes the performance of Mondrix executing on the bochs [Sou03] machine simulator. We present a Linux kernel bug that MMP exposed, and then explain our evaluation methodology. Section 8.3 explains the limits of boch's accuracy, and Section 8.4 presents the results of our experiments.

The simulator reports a slowdown, relative to an unmodified kernel, of less than 12%, according to a simple performance model. For all benchmarks, the MMP protection tables occupied less than 11% of the memory used by the kernel during the execution of that benchmark.

8.1 MMP exposes an error

MMP 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 calling 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 task structure (also called the user area) based on the process identifier. The function calls `free_task_struct` on the task it looks up. This function call should not actually free the task structure because the function decrements a reference count that was incremented earlier in the `proc_pid_lookup`. `free_task_struct` only frees the task structure if the structure's reference count is zero. However, 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 MMP memory supervisor revokes all permissions on memory that is freed, the MMP system 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. However

this routine also 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 2.5, and 2.6.0 uses the new system. We did not check if the new code manifests the same bug we found in 2.4.19.

While we only found two bugs (or two instances of a single bug), the MMP system used some of the most generic, well-tested drivers present in Linux. The bochs simulation environment only supports a model for an NE2000 network card and an EIDE disk, Linux support for both of which is simple and stable. Drivers for newer hardware, or higher performance drivers probably have more bugs.

8.2 Experimental methodology

We ran a variety of system-intensive workloads on Mondrix and measured the effect of isolating kernel modules in separate protection domains that share memory via explicit calls to the memory supervisor. We booted Mondrix on the bochs [Sou03] machine simulator, which models hardware in sufficient detail to boot and run Linux. We added a functional model (with no performance model) of the MMP hardware to bochs, and booted Mondrix on our modified version of bochs. The MMP memory supervisor in Mondrix manages the MMP hardware modeled in bochs. This setup is complete enough to check all data accesses and instruction fetches to verify that the domain performing the action has proper permissions.

Module	**Description**
kernel	Most of the Linux operating system. This is the program image decompressed by the boot loader.
printk	This is an ad-hoc collection of the kernel functions and data consisting of `printk` and related functions (e.g., `sprintf`, vsprintf).
mmp-kernel-symbols	This is a special module used to give kernel information about its own symbol table.
ide-mod ide-disk ide-probe-mod	Collectively, these are the EIDE disk driver.
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.
unix	Unix domain sockets (used by syslogd).

Table 8.1: The names and descriptions of the modules that Mondrix loads.

The OS was booted fresh before each trail. All utilities were from the Debian Linux distribution as of November, 2003.

The code in Mondrix is divided into protection domains as described by Table 8.1 (a picture of the domain structure appears in Figure 7-1). Most of the protection domains hold kernel modules, but domain 1 holds most of the kernel, and domain 2 holds the collection of kernel functions that print, write and format strings. The code in each protection domain must explicitly share memory with code from any other domain.

Benchmark	Description
boot-shut	Boot and shutdown the operating system.
download	Download a 46MB file from a web server (running on the same machine) using HTTP with the `curl` utility.
find	find /usr -print — xargs grep kangaroo; /usr is 183 MB, 912 directories with 10,657 files.
apt	apt-get remove -y gcc gdb automake make perl; apt-get install -y gcc gdb automake make perl; These commands remove, and then download and install 5 Debian packages totaling 5.7 MB in size.
config-xemacs	./configure for xemacs 21.4.14

Table 8.2: The names and descriptions of the benchmarks run by Mondrix to evaluate MMP support in the Linux kernel

Table 8.2 shows the benchmarks we used to evaluate the Linux prototype. They are common tasks that Linux users perform often, so users care about their performance. Also, most of the benchmarks stress the disk and network subsystems of Mondrix. Mondrix isolates the drivers for these devices in their own protection domain, and includes kernel code to manage permissions on the memory that holds disk and network data as it goes through its life cycle of being read, buffered, accessed, and discarded by the kernel. These workloads, and the accuracy of our simulation framework allow us to evaluate the performance costs of adapting an application (in this case the Linux kernel) to use MMP.

The boot and shutdown is not a representative workload, because the MMP-enabled boot was not optimized, and it is slow (symbol lists are dynamically sorted when they could have been sorted statically). It is provided for completeness. The download benchmark stresses both the disk and network code as a web server reads files off disk and sends them across the network to a client. The download benchmark places both the web server and the downloading client on the same simulated machine to avoid problems with network timing described below. The file downloaded is a 44.1KHz, 16-bit wav file, losslessly encoded with `shorten`. [1] The find benchmark is disk and filesystem intensive. The apt benchmark uses the Debian package distribution system to remove and then install several useful software utilities, balancing the use of the network, the disc, and the processor. The configuration of xemacs is a long running test that stresses process creation, deletion and scheduling, and it uses the disc by reading and writing files. It is the only test that does not primarily stress the network or the disc, but it runs for such a long time that the kernel memory allocators reclaim memory.

All workloads ran on the MMP-enabled version of bochs, which checked correctness, generated statistics about the workload, and also generates an address trace. A performance accurate MMP simulator and cache model consumes the address trace and generates statistics.

[1] It is a soundboard recording of the Grateful Dead playing in Austin, Texas on 11/15/71 performing a cool jam between El Paso and Casey Jones.

8.3 Limitations of model accuracy

Our model does not model the interface between the OS and the hardware accurately. The results account for all reads and writes to the protection tables, but the instructions necessary for the software to write the tables are not modeled. However, this inaccuracy is not serious, because the memory supervisor does not write the tables frequently, as the data in Table 8.4 shows.

The way bochs measures time does not allow realistic network communication between the simulated system and the real network. We configured bochs to model a 14MHz processor. But instead of appearing on the network as a slow computer, it appears to be a computer that speeds up and slows down randomly, disrupting TCP (transmission control protocol) traffic. TCP adapts to a computer's ability to process data, sending more data if a computer can handle it, and sending less data if it can't. The erratic behavior of the simulated system as a network client of a non-simulated server sometimes caused TCP connections to become very slow, making experimental results non-repeatable. We measured high variability in the number of instructions needed to transfer data, while the cache miss rate stayed constant, indicating the same processing path was being executed a variable number of times. When the simulated system was a server, persistent device timeouts on transmit made non-simulated clients back off, slowing data transfer and also making experiments unrepeatable. The only benchmark in which the simulated client contacts a non-simulated server is the apt benchmark, which retrieves 5.7 MB from the Debian package servers. This amount of data was small enough to allow repeatable results.

A workload executing on bochs appears to speed up and slow down, because boch's only notion of time for a 14MHz processor is that the execution of 14,000,000 instructions means one second has passed. In reality, different workload characteristics means the simulator may take one second to execute 14,000,000 instructions, but then the instruction mix changes, and the simulator only executes 7,000,000 instructions in the next second. The simulated system appears to have slowed down to the outside world, because it takes two real world seconds to execute the 14,000,000 instructions that bochs counts as one simulated second.

Boch's primitive notion of time frustrated our attempt to have the system running on top of bochs communicate via its simulated network device to non-simulated machines.

8.4 Results

The accurate MMP simulator models a 1-MB 4-way associative cache. The processor performance model is simple. Each instruction takes a cycle to execute, but first level cache misses are free. This is intended to model an out-of-order superscalar processor. In our model, memory takes 60ns to access, and the processor runs at 5GHz. This is intended to represent the performance of an aggressive processor in the next few years. The number of cycles is computed as the number of instructions plus 300 times the number of second level cache misses.

Table 8.3 shows the results of running the benchmarks on Mondrix on bochs, reporting instruction counts, memory references, cache misses, and cycles. In addition to counts, the table reports the percentage increase relative to an unmodified Linux system.

The config-xemacs workload shows an interesting bistability. Sometimes when this benchmark was run, it causes approximately 800,000 cache misses, and sometimes it causes approximately 1.2 million. The variation in time to login to the system, mount the disk

Benchmark	Instrs $\cdot 10^6$		Refs $\cdot 10^6$			$ misses $\cdot 10^6$		Cycles $\cdot 10^6$	
	cnt	**%incr**	**cnt**	**%incr**	**sw/tb**	**cnt**	**%incr**	**cnt**	**%incr**
boot-shut	589	120.4%	315	115.8%	(96%/19%)	2.01	71.3%	1192	92.5%
download	450	12.4%	338	41.5%	(14%/27%)	4.95	2.5%	1935	4.6%
find	609	11.5%	482	35.9%	(13%/23%)	7.76	1.6%	2936	3.5%
apt	513	15.3%	388	41.3%	(18%/23%)	5.49	6.9%	2161	8.8%
config-xemacs	1146	12.0%	821	30.5%	(14%/16%)	1.15	-12.1%	1491	5.3%
config-xemacs2	1140	11.3%	820	33.4%	(17%/17%)	0.80	13.0%	1380	11.6%

Table 8.3: Processor performance data for workloads running with an MMP-enabled Linux kernel. Instruction counts (**Instrs**), memory references (**Refs**), and cache misses from a 1 MB, 4-way associative cache (**$ misses**) are given. The total cycle cost is the instruction count plus 300 times the number of cache misses. The **cnt** column is the count of events, the **%incr** column is the percentage increase of the count over executing the same workload on an unmodified kernel. The **sw/tb** column breaks down the increased memory references between references made by the addition of the MMP software (**sw**), and by references to the permissions tables (**tb**).

with the xemacs workload and run it, determines one of the two operating regimes. Both regimes were observed for Mondrix and Linux, with the low miss regime happening about 25% of the trials. We present both results, where `config-xemacs` is the more probable high-miss regime, and `config-xemacs2` is the less probable, low-miss regime.

The experiments show a surprising uniformity given the different nature of the workloads (discounting the boot, which has not been optimized). The number of additional instructions is between 10%–15%, and additional number of memory references is between 30%–42%. These memory references consist of references made by the MMP memory supervisor, and the references to the permissions tables made by the hardware PLB miss handler. The locality of these references is much higher than the locality of the rest of the kernel since the number of cache misses increased by only -12%–13% (0–7% if discount `config-xemacs`).

According to the detailed MMP simulator, all of our benchmarks experience slowdowns of less than 12%. This number comes close to our design goal of less than 10% slowdown, and is likely to be lower in practice. These performance degradation figures are exaggerated because we show only kernel instructions, discounting user and kernel daemon instructions. If kernel daemons or user programs had greater memory reference locality (and hence MMP table reference locality), the system-wide overhead numbers would be lower. Table references by the PLB miss handler increase the number of memory references made by the kernel by an average of 21%, while Table 5.3, which shows the results of measuring user programs, shows a maximum overhead of 7.5% and an average of 2.9%, indicating that user programs have much more locality than the kernel (a trend noted by other studies [REEBWG95]). We also discount I/O latency, which would be a dominant cost for every benchmark except configuring xemacs, making the user-visible performance impact of MMP negligible.

Table 8.4 shows data about the permissions tables. All of the benchmarks require permissions tables that are less than 11% the size of memory used by the kernel itself. To account for address space sparsity, we compute the amount of memory used by the kernel by counting the number of 4 KB pages with at least one byte in use, and multiplying by 4 KB.

The table also shows that references to the permissions table account for 14%–24% of the total memory references of Mondrix, so permission table references are a limiting factor

Benchmark	Space	X-ref	Upd	ld/lk	Rf/Up	mprots	frees
boot-shut	1.1 MB 2.8%	9.9%	1.6%	2.5	2,050	86,116	53,808
download	1.4 MB 10.7%	23.7%	1.9%	2.6	576	267,470	207,045
find	2.8 MB 5.0%	20.5%	1.6%	2.5	647	355,665	262,072
apt	1.9 MB 4.2%	19.4%	2.8%	2.6	563	315,773	260,748
config-xemacs	1.7 MB 3.6%	14.1%	5.8%	2.5	644	602,212	514,750
config-xemacs2	1.6 MB 3.6%	14.2%	5.7%	2.5	642	602,939	515,443

Table 8.4: Permissions table data for workloads running with an MMP-enabled Linux kernel. The **Space** column shows the absolute size of the permissions tables, and the size as a percentage of used kernel memory. The **X-ref** column divides the number of references to the permission table by the number of memory references made by the MMP-enabled kernel. **Upd** is the percentage of memory traffic to the permissions tables that are writes, **ld/lk** is the number of loads needed to find the proper entry on each permission table lookup. The **Rf/Up** column gives the average number of memory references between updates to the permissions table (mprots or frees). The **mprots** and **frees** columns indicate the count of calls to allocate and release memory permissions (including those made via memory allocation).

for system performance. As noted above, these references have much better locality than the Linux kernel does in general, which mitigates their performance impact.

Table 8.4 also shows that a great majority of table references (95%+) are loads, indicating that the cost of table updates is acceptably low. The number of loads per table lookup is lower than the figure reported for user programs in Table 5.3. In Table 8.4, the number is around 2.5 indicating that about half of the memory regions in use are page-sized or larger. For the user-level programs using fine-grained protection, this figure was closer to 2.9 indicating almost exclusive use of leaf-level tables. Finally, the number of references between updates to the permissions table is very small, about six hundred. This number is much lower and more consistent than the results for user-level programs in Table 5.1, because the OS is making use of different domains, and is actively managing permissions.

Benchmark	Intr	Intr%	Sched	Proc	U/K
boot-shut	1,111,546	6.6%	1,075	264	66%/34%
download	3,696,481	29.4%	5,606	7	51%/49%
find	6,790,923	39.5%	3,768	26	60%/40%
apt	3,121,677	25.6%	20,447	154	83%/17%
config-xemacs	1,707,515	4.9%	12,989	3418	89%/11%
config-xemacs2	1,570,121	4.5%	13,828	3418	89%/11%

Table 8.5: OS characterization of workloads running on Mondrix. The **Intr** column shows the number of interrupts, while **Intr%** shows the percentage of instructions executed in the servicing of interrupts. The **Sched** column lists the number of scheduling events (calls to `sched`). The **Proc** column lists the number of processes that exited during the benchmark. The **U/K** column shows the percentage of time spent in user code versus kernel code.

Table 8.5 gives a characterization of the different workloads in terms of the operating system services they request. Download, find and apt spend at least a quarter of their time processing device interrupts for the disk and network. Apt and configuring xemacs

cause a lot of process scheduling; the apt workload writes and unpacks many files to disk, causing many process suspensions, while config-xemacs creates and kills over three thousand processes. All of the workloads besides config-xemacs, and to some extent apt, spend a significant period of time in the kernel.

Benchmark	**MMP**	**Free**	**Grp**	**Kern**
boot-shut	8.7%	1.3%	22.7%	87.6%
download	4.0%	2.1%	0.4%	5.8%
find	5.0%	2.1%	2.3%	2.2%
apt	6.0%	2.5%	1.6%	5.2%
config-xemacs	4.1%	2.9%	3.2%	1.8%
config-xemacs2	4.1%	2.9%	3.1%	1.2%

Table 8.6: Breakdown of instruction overheads for workloads running on Mondrix. The columns attribute the extra instructions caused by MMP support to the following mutually exclusive catagories: **MMP** the top half of the MMP memory supervisor; **Free** instructions spent in **mmp_mem_free**; **Grp** group protection domain code; and **Kern** kernel modifications to call MMP routines. The total of these four columns is the total percentage of extra instructions.

Table 8.6 attributes the extra instructions to various sources in the memory supervisor and in the kernel. Instructions added to the kernel to manage memory permissions account for less than 6% of the additional instructions. Group domain membership and the checks that must occur when freeing memory are the two largest costs within the memory supervisor. The overheads sum to the instruction overhead presented in Table 8.3.

Benchmark	**Scar**	**PLB**	**PLB_lcl**	**PC sc**	**GPLB**
boot-shut	7.9%	4.0%	58.8%	0.2%	0.3%
download	13.6%	9.1%	62.6%	0.7%	0.3%
find	14.1%	8.0%	48.7%	1.7%	0.6%
apt	11.9%	7.4%	58.4%	0.5%	0.2%
config-xemacs	9.4%	5.2%	55.8%	0.1%	0.5%
config-xemacs2	9.3%	5.3%	56.8%	0.1%	0.1%

Table 8.7: Performance data for the MMP permissions caching hardware for workloads running on Mondrix. The **Scar** column gives the miss rate of the address register sidecars, the **PLB** column the global miss rate of the PLB, the **PLB_lcl** column the local miss rate of the PLB. The **PC sc** column is the percentage instruction fetches that cause the program counter's sidecar to miss. The **GPLB** column is the miss rate of a 512 entry, 4-way associative cache of switch and return gates.

Table 8.7 shows the performance of the hardware on-chip caching structures. The sidecars were effective, probably due to the locality of the stack and string operations on the x86. The PLB's local miss rate is commensurate with second level caches. The table also shows that the number of misses from the PC sidecar is very low. This reinforces our decision to encode the gate permissions in their own table. If they shared the standard permissions table, then any transfer of control across a gate value would result in a PC sidecar miss, greatly increasing the PLB traffic from the instruction stream. The miss rate from the gate PLB is low, because it holds a large fraction of the total gates in Mondrix.

8.5 Evaluation of cross-domain calling in the Linux kernel

Benchmark	Count $\cdot 10^6$	Inst/Call	I/C cross-dom	Self/Other	
boot-shut	1.0	623	1,226	49%	51%
dnld	2.5	185	345	46%	54%
find	4.3	145	218	34%	66%
apt	2.7	198	493	60%	40%
config-xemacs	2.6	448	5,171	91%	9%
config-xemacs2	2.6	458	6,652	93%	7%

Table 8.8: Cross-domain calling behavior for workloads running with an MMP-enabled Linux kernel. The **Count** column is the number of cross-domain calls in millions. The **Inst/Call** is the average number of instructions between cross-domain calls, while **I/C cross-dom** is the number of instructions between cross-domain calls that actually cause a domain change. The **Self/Other** column indicates the percentage of cross-domain calls that a domain makes to itself versus those that cause a domain change.

Table 8.8 summarizes cross-domain calls in Mondrix. The denominator for all of these experiments is the number of instructions executed in the kernel (user/kernel split is in Table 8.5). The protection domain granularity enforced by MMP is very fine-grained. The table shows that cross-domain calls are frequent, and many are cross-domain calls from a domain to itself. A domain makes cross-domain calls to a function that it exports to another domain (Section 4.3.1). In these experiments the top half of the memory supervisor and the kernel were in the same protection domain, so calls between them (which are frequent) are counted as a cross-domain call from a domain to itself. For the benchmarks that stress the operating system, there are only hundreds of instructions between cross-domain calls. For the `./configure xemacs` workload, which stresses process scheduling more than system services, there are 5,000–6,000 instructions between cross-domain calls.

During all of these benchmarks, the cross-domain call stack doesn't grow deeper than 64 entries, indicating that the active part of the cross-domain call stack will easily fit in the processor data cache, and will not cause additional memory traffic.

The permissions tables of Mondrix consumed less than 11% additional memory space, and according to a simple performance model, Mondrix ran within 11% of the performance of an unmodified Linux. Cross-domain calling is very frequent in Mondrix, as much as a call for every few hundred instructions for some benchmarks. This frequency justifies architectural support.

Chapter 9

Enforcing Stack Permissions

This chapter presents a design for adding protection of stack memory to MMP. Stack storage can not be managed by MMP's protection domain mechanism, so this chapter introduces new hardware mechanisms for stack protection, which are similar to other MMP structures.

Stack permissions are associated with a thread. Stack state is a form of thread-local storage, while heap data is generally used for a software service. Threads travel between services (and thus protection domains), referencing the state they store on the stack. MMP's protection domain mechanism is insufficient for stack memory, because permissions granted to a protection domain are granted to all threads in the domain. Using the protection domain mechanism for stack memory would allow multiple threads in the same protection domain to access each other's stack frames.

Stack sharing is difficult to implement in a system that provides hard module boundaries, but it is necessary to preserve the standard function call model for cross-domain calls, and supporting stack-allocated data structures increases performance.

Section 9.1 explains why and how the memory supervisor manages stack memory for all domains. We then present two new hardware mechanisms to provide access permissions for the stack. The first (in Section 9.2) is a set of three registers that provide a fast activation frame. The second mechanism (in Section 9.3) is a simple, thread-local permissions table, which distinguishes writable words on the stack frame. The chapter concludes with alternatives to sharing stack memory.

9.1 Memory supervisor's stack responsibilities

Because stacks are used by threads which move between protection domains, no domain has a unique claim to be the stack owner. We resolve this dilemma by making the memory supervisor the owner of the stack; the supervisor allocates it, owns it, and manages it. The supervisor allocates stack segments using the function `mmp_alloc_stack(len)`. This call returns a pointer to the stack segment, and the supervisor records the location of the segment and the domain of the code that created it.

Stack permissions are thread-local, so the supervisor needs to be aware of what thread is running. When a thread is scheduled on a CPU, the thread manager must make the supervisor call `mmp_set_stack(stack_seg, cpuid)` to tell it that a certain stack is now

active on a certain CPU. The supervisor checks that this thread manager is from the same domain as the one that created the stack. When the supervisor receives a call to set stack permissions, it checks that the request is for the active stack.

9.2 Managing stack permissions with extra registers

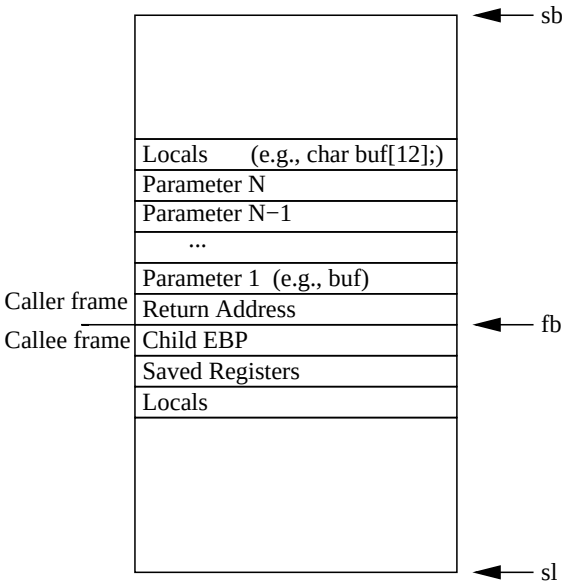


Figure 9-1: Providing stack isolation with three hardware registers. Register `sb` (stack base) indicates the base of the stack, and `sl` (stack limit) indicates its limit (stacks grow down). The `fb` (frame base) register indicates where the current frame begins.

Establishing an activation frame should be fast, and the permissions for reading and writing the frame should be local to the currently executing thread. We introduce two thread-local registers to enable the fast establishment of an activation frame. We add a third to provide a simple and efficient mechanism to get read-only access to previous stack frames.

Figure 9-1 shows the layout for the stack used on the x86 by both Windows compilers and `gcc`. We separate the caller's frame from the callee's frame. On the right are pictured the pointers stored in three thread-local registers.

When the kernel schedules a process, it calls into the supervisor to activate the stack for that process. The supervisor also fills in two registers for that thread—frame base **fb**, and stack limit **sl**. These registers demarcate a read-write region for the currently executing thread. The hardware allows reads and writes to addresses between **sl** and **fb** (stacks grow down so **sl** $\leq$ **fb**). The **fb** value points to the base of the current activation frame. The supervisor manages the save and restore of these registers. The supervisor allocates the stack for a given thread, so it can initialize the stack limit register and validate the frame base register.

On a cross-domain call, the processor saves the current value of **fb** to the cross-domain call stack, and it copies the current stack pointer into **fb**. The processor checks that the new **fb** value is smaller than the old value, insuring that on cross-domain calls, **fb** grows down,

but not below **sl**. The memory supervisor insures that when a thread starts executing, **fb** points within the stack memory for that thread. Since cross-domain returns can only set **fb** to a value that was checked by either the supervisor or the processor, these mechanisms ensure that **fb** always points within stack memory.

The supervisor uses the stack base register (**sb**) to mark the region between **fb** and **sb** with read-only permissions. **sb** is not essential for correctness, because the **fb** and **sl** pair provide a writable frame, and the table described in the next Section provides read permission on previous frames. However checking a table is more complicated than checking the base and bounds of two registers, so the stack base register is a useful optimization. The memory supervisor initializes **sb** using its knowledge of the size and location of the stack. The supervisor saves and restores **sb** along with **fb** and **sl** in storage specific to a given thread.

9.3 Stack allocated parameters

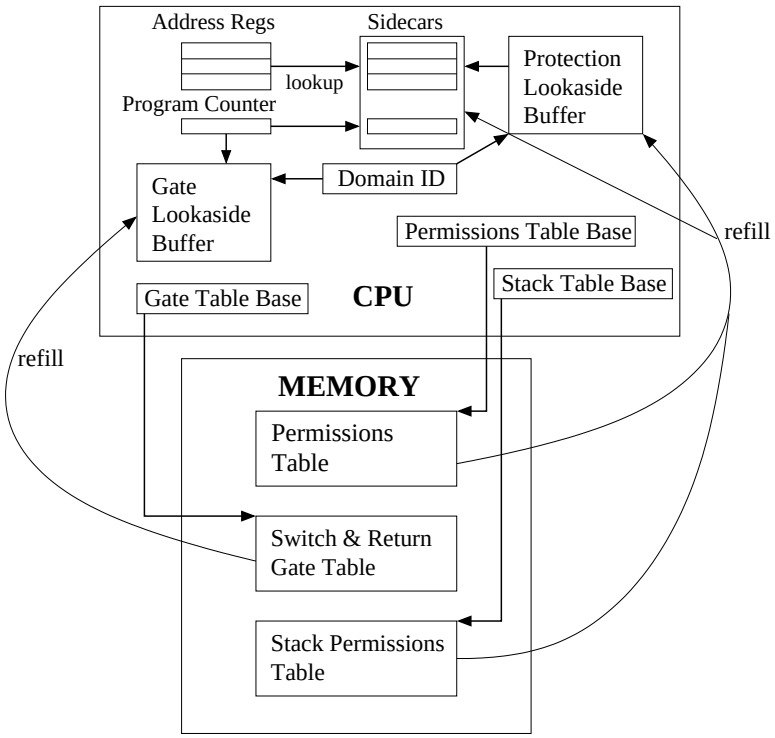


Figure 9-2: The major components of the Mondriaan memory protection, including support for managing stack permissions.

A thread-local table of 1 bit per word provides a mechanism to allow a thread to write into a previous stack frame. The three additional registers we propose provide a writable stack frame, and allow a thread to read previous stack frames, but they do not provide write permissions on previous frames. For each word in the stack, if its corresponding bit is set, the word is writable and readable, otherwise it is only readable. This design does not allow inaccessible words on the stack, as that would require more bits.

Figure 9-2 is a more detailed version of Figure 4-1 that includes the stack permissions

table. The stack permissions table is a thread-local table (not a domain-local table), which allows a thread to pass permissions to successive frames of stack memory, independent from the domain in which a thread executes. Like the gate permissions table, it is small and specialized. Unlike the gate permissions table, it is cached in the main PLB.

As an example of how a thread can use the stack permissions table, consider a thread that will make a cross-domain call to `mmp_get_my_ro_section`, which is a memory supervisor function (whose prototype is in Appendix A). The thread can make two stack-allocated long integers writable, and then pass pointers to them to `mmp_get_my_ro_section`. When the thread enters the memory supervisor domain, it still has write permission on those two words because they are marked in the stack permissions table. When the OS deschedules the thread, it notifies the memory supervisor which revokes the threads stack permissions by flushing the stack address range from the PLB and changing the stack permissions table pointer.

This model of stack permissions lets a thread manipulate the stack independently from its protection domain, which might not always be desirable for a given domain. For instance, consider the case where domain `A` calls domain `B` which calls domain `C`. Domain `A` can't export stack permissions to domain `C`, without trusting `B` to not revoke the permissions. A thread can add or revoke stack permissions in any domain. However, parameters that a domain must control can always be allocated from the heap.

9.4 Alternatives to sharing stack memory

Designs for protected stack sharing are rare for several reasons: it is difficult for a system to provide firm module boundaries while allowing sharing of stack memory; using stack-allocated memory is not necessary for a computer system; and switching stacks is simple for both hardware and software.

Capability based systems, where capabilities were used for fine-grained memory protection (e.g., the Cambridge CAP [WN79]), do not provide stack storage for cross-domain calls. Storage is heap-allocated and capabilities to the storage are passed in the call. Call gates on Intel's x86 architecture [Int96] cause the processor to switch stacks and to copy parameters from one stack to another. Even modern systems intended for inter-domain safety, like Nooks [SBL03] copy stack parameters and switch stacks in software, disallowing stack sharing.

An MMP system can use the stack switching techniques of previous systems. Two MMP domains can switch stacks before or after a cross-domain call in software. The caller might have permissions to establish an initial stack state, which is verified (or trusted) by the callee.

Chapter 10

Adding Translation to MMP

Up to this point, we have presented Mondriaan memory protection as a memory protection system, but it can do more. MMP effectively associates metadata with address ranges, and the flexibility of the MMP permissions tables allows for additional metadata, along with permissions information. Simple, efficient hardware can interpret the additional metadata, just as the MMP hardware explained in this thesis efficiently interprets permissions metadata. Adding semantics to memory by additional metadata is done in many systems, for example multi-processor systems keep metadata about the sharing state of cache lines in a directory data structure which is read by hardware [LLG$^+$90, CKA91, HLH92].

The motivating example for additional MMP metadata is zero-copy networking. We use the term, "zero-copy networking" to refer to any technique which tries to reduce data copies during the processing path which moves user data to or from the network. Section 10.1 provides background on zero-copy networking. MMP's approach to zero-copy networking maintains backwards compatibility with current networking code, specifically the use of the `read` system call.

Byte-level translation, along with MMP's word-level permissions, allow an operating system to implement zero-copy networking. The OS uses byte-level translation to map byte-aligned network packet payloads from different packets into a contiguous, user-supplied buffer (provided as an argument to the `read` system call). We call the extended system MMPT, for Mondriaan memory protection with translation. Because non-permissions information is stored in the "permissions" table, this section will refer simply to tables, not permissions tables.

Section 10.3 describes MMPT's byte-level translation. Section 10.3 describes how an OS would use MMPT to implement zero-copy networking, while Section 10.4 describes the processor hardware that interprets the translation data. Section 10.5 discusses the complications to the processor pipeline caused by adding byte-level translation, and how to address them. It also presents the table representation for translation information. Section 10.6 concludes with an evaluation of MMPT for simulated zero-copy networking.

10.1 Zero-copy networking background

There are many proposals in the literature for zero-copy networking [Chu96, PDZ00, vEBBV95]. Most are successful at eliminating extra copies within the kernel. The hardest implementation issue is eliminating the copy between the kernel and the user. Systems like IOLite [PDZ00] change the user/kernel interface and programming model to pass around collections of pointers. The user is aware that her data is split into various memory regions, which complicates programming. Another approach lets user handlers manage the copy from the network interface directly [MKF+98]. Direct access to the network interface requires special hardware, does not interact well with multi-programming and demand paging, and results in the entire packet, not just the payload, being transfered to user space.

A final approach [Chu96] uses page remapping, which can be implemented under the standard `read` system call. The implementation in [Chu96] is for ATM networks where the maximum transfer unit (MTU) is greater than a 4 KB hardware page size. Since it uses page remapping techniques, it is limited to the hardware page granularity. The largest standard Ethernet packet is less than 1512 bytes, which is smaller than most modern page sizes.

We believe the remapping approach is the best for zero-copy networking, and MMPT eliminates the page size restriction and extends the approach to data that is split among multiple packets or packet fragments. It offers the programming ease, and backwards compatibility of linear buffers with the performance of zero-copy networking stacks.

10.2 Memory translation

To support address translation, the processor stores a translation offset for each address register, and adds the contents of the translation register to the effective address calculation for loads and stores.

Figure 10-1 shows two translated memory regions in a protection domain. When code in the domain accesses addresses in the range `0x80002800`–`0x800028FF`, it can read and write that memory. When code accesses addresses in the range `0x1200`–`0x12FF`, the processor restricts its access to read-only, and adds `0x80001600` to the address. So accesses to address range `0x1200-0x12FF` actually load data from the address range `0x80002800`–`0x800028FF`. Different memory regions can have different translation values, so memory that is discontiguous can be translated so that its images are contiguous, as shown in the figure.

10.3 Implementing zero-copy networking with MMPT

Figure 10-2 shows how translation can implement zero-copy networking with a standard `read` system call interface. The kernel buffers packets as they arrive on a TCP connection. It then maps the payload from these packets into contiguous segments (provided by `read`) which the user can then access. Permissions are only given for access to the data payload so the network stack is isolated from a malicious or buggy user.

In Figure 10-2, the client domain passes a buffer (a 3 KB buffer occupying the address range `0x1000`–`0x12FFF`) to the kernel via `read`. The kernel becomes the owner of the

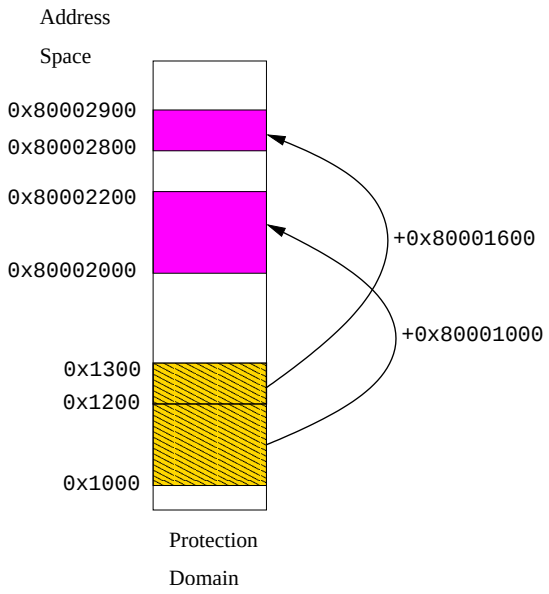


Figure 10-1: An example of byte-level translation. Two buffers, starting at `0x80002000` and `0x80002800` are translated down to start at `0x1000` and `0x1200`, respectively. Code in the protection domain can read and write the buffers in high memory, but can only read the buffer's translated image in low memory. The arrow from low memory to high is labeled with the address offset that translates the low buffer to the high.

buffer, and it remaps the packet payloads into the buffer without copying them. When the user reads the buffer (e.g., `0x1000`), the processor adds the translation offset so the data comes from `0x80002000` which is where the packet payload resides. The translation is represented by two user segments which have translation information, i.e., `<0x1000, 0x200, RO, +0x80001000>`, and `<0x1200, 0x100, RO, +0x80001600>`. The final segment field holds the translation offset. The packet headers are not translated into the client domain. The client never sees the packet headers.

Segment translation does not preclude other levels of memory translation. For an embedded system that uses a physical address space, segment translation could be the only level of memory translation in the system. For a system that uses virtual addresses, the result of segment translation is a virtual address which is translated to a physical address by another mechanism. Translations are not recursive, a translated segment cannot be the target of other translations.

The MMP system does not dictate policy, but one reasonable choice is that only the protection domain that owns a segment can install a translation, and the translation must point to another segment owned by the same protection domain. This property would be checked by the supervisor when it is called to establish the mappings.

10.4 Translation hardware implementation

MMPT requires sidecar registers, storing a translation offset in the the address sidecar register as shown in Figure 10-3. The processor adds the translation offset to every memory address calculation, allowing a region of memory to appear to reside in a different address

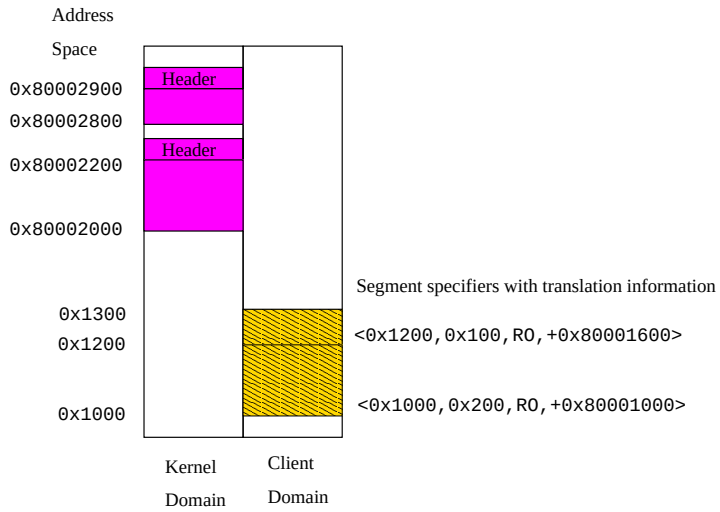


Figure 10-2: Using memory protection and segment translation to implement zero-copy networking. The network interface card uses DMA to copy packets into the kernel. The kernel exports the packets to an untrusted client by creating segments for the payload of the packets. Segment translation presents the illusion to the client that the packet payloads are contiguous in memory at `0x1000-0x12FF`.

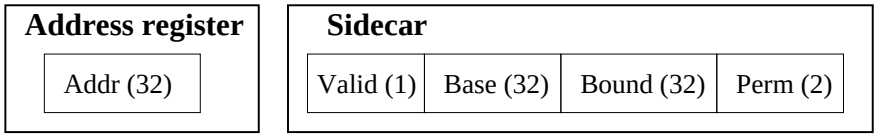


Figure 10-3: The layout of an address register with sidecar which has translation information (shaded portion).

range. This will increase the typical two operand add used for address arithmetic to a three operand add. The additional 3:2 carry-save adder will add a few gate delays to memory access latency.

Segment translation does not cause cache hardware aliasing problems, because translation occurs before the access is sent to the cache and memory system. In the example in Figure 10-1, no data is cached for address `0x1000`, because data is never fetched from that address, it is fetched from `0x80002000`. This can create a software pointer aliasing problem if software assumes that only numerically equal pointers point to the same data. In the example, `0x1000` and `0x80002000` point to the same memory, but they are not numerically equal. Software is often written to assume that pointers which are numerically unequal do not point to the same memory.

Since all memory meta-data is changed via supervisor calls, the supervisor can enforce policies that mitigate the negative effects of software pointer aliasing. One policy would be that, since a domain must own both the translated segment and its image, the domain can only export the segment, and not the image. This prevents other domains from seeing the translation and becoming confused, but would support applications like zero-copy networking.

10.5 Complications from byte-level translation

Byte-level translation interacts with multi-byte loads and stores to create two architectural complications. The first is that addresses which appear to be aligned can create unaligned references when used. The address issued by the processor is the user address plus the translation offset. If a segment is translated to an odd-byte boundary (e.g., `<0x1000, 0x200, +0x80002003>`), then a reference to user address `0x1000` becomes an unaligned reference to `0x80003003`. Some modern processors can handle unaligned loads from the same cache line in a single cycle, but require two cycles for unaligned loads that cross cache line boundaries.

The second issue arises when a single multi-byte load crosses translation boundaries. Returning to the example in Figure 10-2, consider the case where the first packet has one fewer byte of data payload: `0x1FF` bytes instead of `0x200`. We can almost represent this situation with the segments `<0x1000, 0x1FF, RO, +0x80001000>` and `<0x11FF, 0x101, RO +0x80001601>`, but the length of our segments and their base address must be word aligned, because the entire table is word-aligned. The problem is with the word at address `0x11FC`. The first three bytes must come from the first segment, and the last byte must come from the second segment.

We call a word that spans segment translation boundaries a *seamed word*. The permissions table must represent seamed words. To simplify their representation, they are defined to be single word segments that must occur on the first word of two adjacent segments, e.g., the word at address `0x11FC` in our example. With this restriction, an entry needs only a single bit to represent that two adjacent table segments have a seam between them, and then two bits to indicate how many bytes of the seamed word come from the first segment. Any multi-byte quantity can be seamed, and can use the word-oriented table representation. For instance, an 8-byte load can have a seam in the first word, or the second.

A seamed load requires the processor to collect the bytes within a single word load from different addresses. Fortunately, the pipeline mechanism is almost identical to what is needed for unaligned loads that cross cache line boundaries—bytes from different locations must be shifted and muxed together. The only difference with seamed loads is that the two locations being read are not within three bytes of each other.

To simplify the hardware, we impose a restriction that seamed stores are not supported. Allowing a store to be broken into two separate pieces located at very different cache addresses can cause severe complications for SMP cache coherence.

10.5.1 Adding translation to sorted segment table entries

Figure 10-4 shows the records in a sorted segment table with translation information. The SST's simplicity allows translation information to be concatenated onto the segment record. There are 32 bits of translation for each segment, with a bit to indicate if the segment has a seam, and two bits to indicate after how many bytes in the last word of the segment does the seam cross into the next segment.

10.5.2 Adding translation to run-length encoded entries

Figure 10-5 shows the run-length encoded record that represents seamed words and translation information. The record is six words long and is pointed to by a table entry which is

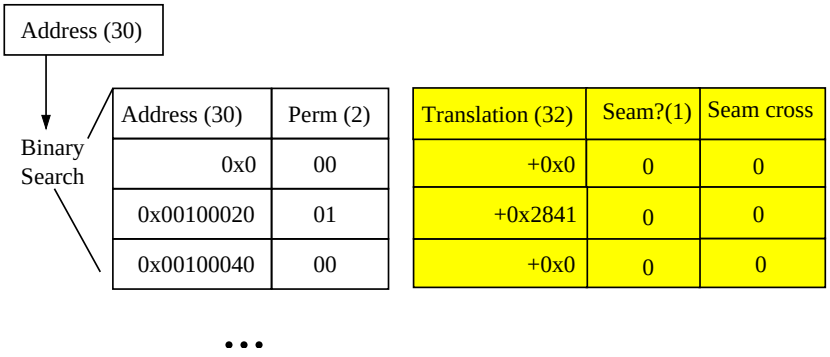


Figure 10-4: A sorted segment table (SST) with translation information. Entries are kept in sorted order and binary searched on lookup.

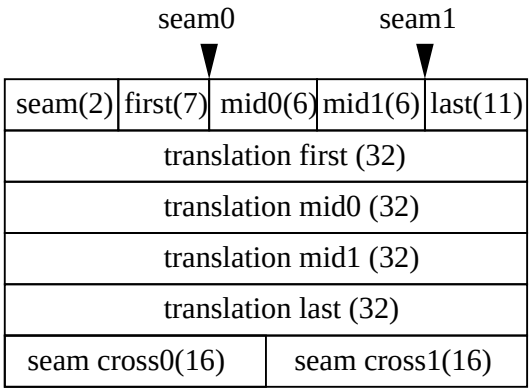


Figure 10-5: The format for a record with a run-length encoded entry and translation information.

a type `10` pointer (see Table 3.2). There are 32 bits of translation for each segment.

The upper two bits of the RLE, used for type information when the RLE entry appears in a non-leaf position, are reallocated in this record to indicate the location of seamed words. The type bits are available, because the translation record is pointed to by the type `10` pointer, which only points to translation records. The arrow heads indicate where seamed words are allowed to occur. The bits are independent and if the first bit (`seam0`) is set, there is a seam between table segments `first` and `mid0`. If the second bit is set (`seam1`), there is a seam is between `mid1` and `last`. The last word of the record contains two 16-bit fields. Each represents the byte cross-over point for the corresponding seamed entry. In the example given above, the cross-over point is 3 bytes because `0x11FC`–`0x11FE` come from the first segment and `0x11FF` comes from the second segment. While only 4 bits of `seam_cross` information is needed (two bits for each seam), the format reserves an entire word to keep the records word aligned.

This record format restricts the system to two seamed entries in every 16 words, and requires that translated segments be representable by a run-length encoded entry. If there are many small regions (e.g., many small network packets or packet fragments) it is better to copy the contents rather than construct many translated or seamed regions.

All fields of the record type are not always needed. The initial word in the run-length encoded entry establishes how many segments there are and if there are any seams. This

determines how many additional words are necessary. Records in the MMPT prototype are fixed length for simplicity, but they could be variable sized to reduce space consumption.

PLB entries in an MMPT system must be at least six words long, to accommodate the translation record.

10.6 Evaluation

We recorded a web client receiving 500 KB of packets and simulated the action of a kernel driver which accepts the packets into kernel memory and then translates the packet payload segments into a contiguous segment provided by the client via the `read` system call. The client then streams through the entire payload. In this scenario, the kernel reads the packet headers, and writes the permissions tables to establish the translation information. The client reads the data, causing the system to read the translation and permissions data from the protection table. This experiment used only the MMP table simulator (as in Chapter 5), and did not include Mondrix—the actions of the OS were simulated by directly accessing the MMP tables.

We compare the number of memory references required for the segment translation solution with the number of memory references required for the standard copying implementation. In the copying implementation the kernel reads the headers, and then reads the packet payloads and writes them to a new buffer. The client streams through the new buffer.

Zero-copy networking saves 52% of the memory refereces of a traditional copying implementation. It has a size overhead of 29.6% for the permission tables. 61% of that 29.6% overhead is for permissions tables and the remaining 39% is for the translation records. 11% of the references are unaligned and cross cache line boundaries. 0.5% of the references are seamed. If we charge 2 cycles for the unaligned loads that cross cache line boundaries, 10 cycles for the seamed loads and discount all other instructions, the translation implementation still saves 46% of the reference time of a copying implementation.

Chapter 11

Additional Applications, Future Work, and Conclusions

This chapter examines other applications of MMP, besides isolating modules. We also present additional applications for the byte-level translation extension presented in the last chapter. The chapter continues with a discussion of how programming languages interact with MMP. Finally, we conclude.

11.1 Additional applications for MMP

We believe that fine-grained protection offers exciting opportunities for application developers. Appel [AL91] surveys some applications that make use of page-based virtual memory. Many of these applications could perform better with finer grain protection and with efficient cross-domain calls.

Fine-grained protection can provide support for efficient array bounds checking. Bounds checking is useful for program debugging and if implemented by MMP would be available to the kernel.

Unsafe languages allow buffer overruns, in stack-allocated and heap-allocated memory, which are a common source of security holes [WFBA00]. A memory allocator could use MMP to place inaccessible guard words between heap allocation units which would catch a program's attempt to write off the end of a piece of memory. A compiler or run-time system could place an inaccessible guard word between local parameters and a procedure's return address, so any attempt to overwrite the return address by overflowing a stack-allocated data structure generates a fault.

A related application, data watchpoints [Wah92], can be easily implemented with MMP. A data watchpoint generates a trap when the processor stores a value into a given word in memory. Some processors support a handful of watched memory locations [KH92, Int97], but MMP scales to allow any number of individually protected words.

MMP can eliminate data copying between the user and kernel, increasing performance. The kernel's address space is usually inaccessible to user code, but some of it could be made writable to the user. MMP would ensure that user code could only write into the data

portions of a kernel data structure, and could not overwrite (or even read) pointers and other sensitive data. Allowing the user to write directly into kernel memory would allow the kernel to avoid copying the user's data on system calls and it would ensure that the data was memory resident (if the kernel had the user write into pinned memory). The kernel interface should not be tied to the details of the layout of a kernel data structure, since the layout might change, but a standard interface could be implemented by direct copying from user to kernel address space.

Generational garbage collectors [LH83] segregate objects into two classes, "new" and "old", where new objects are garbage collected more frequently than old objects. The runtime system relies on the invariant that old objects do not point to new objects. Checking in software that updates to old objects do not cause them to point to new objects is time consuming. The runtime system could use MMP to write protect all pointers in all old objects. Whenever the program writes one of these locations, the runtime system is notified by the memory supervisor which handles the protection fault. The runtime system checks the newly written pointer to determine if an old object is being updated to point to a new object. If it is, the new object (and everything it points to) is now considered old, and the runtime system has preserved the invariant that old objects do not point to new objects.

Infiniband[Ass01] is a high-performance switched interconnect architecture for processors and I/O devices. Infiniband supports Remote Direct Memory Access (RDMA) operations where the initiator of the operation specifies both the source and destination of a data transfer as memory addresses, resulting in zero-copy data transfers with minimum involvement of the CPUs. MMP could be used in a processor or I/O device to regulate access to these remote memory windows.

Flexible sub-page protection enables distributed shared memory systems like Shasta [SGT96] and its predecessor [SFL+94]. Shasta found significant benefit from breaking up different data structures into different sized chunks to manage consistency. But since Shasta's line sizes did not map to virtual address pages, it performed access checks in software. While the authors of Shasta used impressive compiler techniques to reduce the cost of these software access checks, MMP would reduce this cost further.

The C region library [JKW95] also allows programmers to maintain coherence for arbitrary-sized data regions, and so would benefit from MMP's ability to mark such data regions as inaccessible if they were owned exclusively by another processor.

Fine-grained memory protection enables security-oriented software to make stronger claims about information flow, e.g., that a particular memory word was never read. Having a "no access" permission value is important for this application.

11.1.1 Combining fine-grained protection and translation

Combining fine-grained protection with byte-level translation is useful for applications other then zero-copy networking. For instance, we can use it to implement stacks efficiently in a thread package. A persistent problem for supporting large numbers of user threads is the space occupied by each thread's stack [GN96]. Each thread needs enough stack to operate, but reserving too much stack space wastes memory. With paged virtual memory, stack memory must be allocated in page sized chunks. This strategy requires a lot of physical memory to support many threads, even though most threads don't need a page worth of stack space. With MMP segment translation, the kernel can start a thread and only translate a very small part of its stack (e.g., 128 bytes). If the thread uses more stack

memory, the kernel can translate the next region of the stack to a segment non-contiguous with the first, so the stack only occupies about as much physical memory as it is using, and that memory does not have to be physically contiguous.

Similarly, if a system is low on physical memory, it can use MMP translation to map non-contiguous memory regions into a more usable contiguous chunk.

Segment translation is also useful because it allows data structures to be linked without a pointer. If **A** has a pointer to **B**, that pointer must be loaded in order to find **B**. If **A** and **B** are translated to be contiguous, then indexing off the end of **A** will find **B**. The address computation of translation replaces the memory reference of the pointer load. This approach would eliminate the latency of loading and dereferencing the pointer, and might reduce memory traffic.

A common data structure that MMP protection and translation could optimize is the mostly-read-only data structure. An example comes from the widely-used NS network simulator [ns00]. Each packet has mostly read-only data. When simulating a wireless network, packets are "broadcast" to nodes, which read the read-only data, but also write a small node-specific scratch area in the packet (e.g., to fill in the receive power which is node specific). The current NS simulator supports this data structure by making a copy of the packet for each node, so that each node has its own scratch area. This copying reduces the size of simulations that are possible with a given amount of physical memory, and takes cycles that could be used for computation. Splitting the packet into read-only and read-write sections and managing them separately is possible, but it complicates a core data structure. By using fine-grained MMP translation, a single read-only payload can be made visible at different addresses within multiple protection domains. Each domain can then have a private read/write region allocated contiguously to the read-only view. This solution allows the nodes to share the read-only part of each packet, while providing each node with a private scratch area.

11.2 MMP and programming languages

This section discusses how a programming language could present an interface to MMP functionality. It also discusses how an MMP system can support programming language exceptions and continuations.

11.2.1 Language-level interface to MMP

Some language features (extant and imagined) can use MMP for their implementation. C++ name spaces could be implemented as different protection domains. The scope of the name space can be determined statically, and all public functions in the name space would be available. Most of the MMP setup work could be done by the compiler at compile time, and instantiated during program initialization.

For strongly typed languages, the language itself is sufficient for access control, but MMP is useful in these systems to guard against implementation bugs. Class loaders in Java are the mechanism by which a reference to a class name from a running program becomes an executable representation of that class. The JVM's primordial class loader would likely want to load trusted classes into some number of reserved protection domains to protect them against implementation bugs. User-level class loaders already offer a rich

set of methods for obtaining an executable Java class. MMP would be an additional useful tool to help control the trust relationships for the class implementation that is being loaded.

A language could provide a `read-only` type qualifier which could be implemented by MMP.

11.2.2 Implementing exceptions and continuations in MMP

Programming language exceptions can be implemented by several methods [RJ00]. One set of techniques cuts the stack, setting the stack pointer and program counter to the exception address. These techniques make exceptions fast, but penalize non-excepting paths by disallowing register allocation of callee-saved registers. Stack cutting between domains is not possible with switch/return gates. For instance, switch and return gates are insufficient to implement `setjmp/longjmp`, which is a stack cutting exception method.

The other set of exception implementation techniques unwind the stack, which removes the penalty of entering the scope of an exception handler, but requires more work for an exception. It also requires more work for each function call because each call can have multiple return values, and the caller must decide if the function is returning because of an exception condition. Switch and return gates support a stack unwinding implementation of exceptions with interpreted multiple return values. This method is used in the MIT Flex compiler [RAB+03] to implement exceptions in the Java language.

Interpreting multiple return values can hurt performance, and multiple return sites can eliminate the need to interpret abnormal return codes [RJ00]. An abnormal condition returns control, not to the instruction following the call, but a succeeding instruction, which is usually a branch to exception handling code. MMP cross-domain calling can support this idiom by allowing multiple return address in a single cross-domain call record. The processor would have to check each of these addresses on a cross-domain return, and it would have to push and pop a variable number of words for each cross-domain call.

Switch and return gates are not sufficient to implement continuation passing style. A gate type that did not push a record on the cross-domain call stack could be provided, and continuations would use them to denote entry points. The hardware would not check returns.

11.3 Conclusion

MMP finally makes practical the longstanding goal of fine-grained memory protection. It provides fine-grained protection with backwards compatibility for operating systems, ISAs and programming models, and does without confusing new programmer-visible abstractions. Most of the best ideas for system structure that proponents of segmentation or capabilities have proposed can be implemented with MMP in a way that is minimally disruptive to existing software.

Our experience in adapting Linux to use MMP indicates that the programming model provided by the MMP hardware fits naturally with how modern software is designed and written. Designers of large software projects use modular boundaries, and MMP can provide hardware enforcement for the module boundaries that already exist.

Programs grow as complex as programmers, tools, and the hardware substrate allow. Delivering a complex feature set with a reliable program has been a constant challenge of

computing. MMP will help programmers uncover programming errors more quickly during testing, and allow programmers to design their system to continue functioning if one module fails because of an error.

Modularity as a technique to provide system stability has a long history in computer architecture and operating systems. Many techniques to increase the modularity of memory are now in widespread use. We hope that MMP continues the tradition.

Appendix A

Interface file for Mondrix memory supervisor

```
#ifndef _MMP_H
#define _MMP_H

#include <asm/mman.h> // PROT_NONE PROT_READ, PROT_WRITE, PROT_EXECUTE
// Switch domains gate
#define PROT_SGATE  0x8
// Return gate
#define PROT_RGATE 0x10
// Convenience
#define PROT_RW (PROT_READ|PROT_WRITE)

typedef unsigned int pd_id_t;

////////////////////////////////////////////////////////////////////
// Initialization

// Called very early, before kmalloc is safe, from main.c.  It allows
// us to record the allocation of memory for the initial RAM disc in
// arch/i386/kernel/setup.c, and pagetables in arch/i386/kernel/init.c
void mmp_early_init(void);
// Called once kmalloc is safe, from main.c
void mmp_init(void);

////////////////////////////////////////////////////////////////////
// Main memory permissions manipulation interfaces.

// The main interface for setting permissions for myself or other domains.
void mmp_mprot(const void* ptr, unsigned int len, int prot, pd_id_t pd);
// Put an sgate on the function.  If pd == caller's pd, put an rgate
```

```
// on the return.
void mmp_func_gate(const void* func, pd_id_t pd);
// Read protection tables
void mmp_get_prot(const void*, pd_id_t, int*);
// Find the PD that owns this address.  Use code or static data, it
// won't work for dynamically allocated memory.
pd_id_t mmp_code_to_pd(const void* addr);
// This allows ide-disk.c to export its strings.
void mmp_get_my_ro_section(unsigned long* base, unsigned long* len);

//////////////////////////////////////////////////////////////////////
// Memory allocator interface
void mmp_mem_alloc(const void* ptr, unsigned int len);
void mmp_mem_free(const void* ptr, unsigned int len);

//////////////////////////////////////////////////////////////////////
// Protection domain creation
struct mmp_req {
   const void*  ptr;
   unsigned int len;
   int         prot;
   int         steal;
};
pd_id_t mmp_pd_subdivide(struct mmp_req* req_vec[]);
// Protection domain deletion
void mmp_pd_free(pd_id_t, int recursive);

// Module interface
struct module;
void mmp_module_init(struct module* mod);
void mmp_module_free(struct module* mod);

//////////////////////////////////////////////////////////////////////
// Group protection domains.  These are software managed protection
// domains, whose protections are ORed into a real protection domain,
// when that pd joins the group.
// 0 is not a legal value for gpd_id_t.  They start at 1.
// Make a memory group.  It takes an estimate of the number of regions
// in the group.
typedef unsigned int gpd_id_t;
// Create a group
gpd_id_t mmp_gpd_create(const char* name, int nregions);
// Destroy a group
void mmp_gpd_destroy(gpd_id_t);
// Add a memory region to a group domain
int mmp_gpd_export(gpd_id_t, void* ptr, unsigned int len, int prot);
// Delete a memory region from a group domain
int mmp_gpd_unexport(gpd_id_t, void* ptr, unsigned int len);
```

```
// Add a group domain to the currently executing domain
int mmp_gpd_add(gpd_id_t);
// Get rid of a group domain from the currently executing domain
int mmp_gpd_unadd(gpd_id_t);

//////////////////////////////////////////////////////////////////////
// Special function for printk-family functions to get read-write
// permission on a buffer whose length is determined by the supervisor
// reading the printk caller's protection table.
void mmp_printk_rw(void* ptr);
void mmp_printk_done(void* ptr);

//////////////////////////////////////////////////////////////////////
// Names of the domains which house common kernel modules.
extern pd_id_t kern_pd;
extern pd_id_t printk_pd;
extern pd_id_t ide_mod_pd;
extern pd_id_t idedisk_pd;
extern pd_id_t binfmt_pd;
extern pd_id_t rtc_pd;
extern pd_id_t pd_8390;
extern pd_id_t ne_pd;
extern pd_id_t unix_pd;
#endif // _MMP_H
```

Bibliography

[Ado02] Adobe Systems Incorporated. *Adobe PDF Plugin*, 2002. `http://www.adobe.com/`.

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 96–107, 1991.

[Apa03a] Apache Software Foundation. *Apache web server*, 2003. `http://www.apache.org/`.

[Apa03b] Apache Software Foundation. *mod_perl*, 2003. `http://perl.apache.org/`.

[ARM00] ARM Ltd. *ARM940T Technical Reference Manual (Rev 2)*, ARM DDI 0144B 2000.

[ASG97] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 134–145, 1997.

[Ass01] InfiniBand Trade Association. *InfiniBand Specification 1.0a*, June 2001.

[BALL89] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. In *Proceedings of the 12th ACM symposium on operating systems principles (SOSP)*, pages 102–113, Dec. 1989.

[Ber80] Viktors Berstis. Security and protection in the IBM System/38. In *Proceedings of the 7th International Symposium on Computer Architecture*, pages 245–250, May 1980.

[Bon94] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[BSP+95] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gn Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan J. Eggers. Extensibility, safety and performance in the SPIN operating system. In *15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, 1995.

[Bur61] Burroughs Corporation. *The Descriptor—a Definition of the B5000 Information Processing System.*, 1961. `http://www.cs.virginia.edu/brochure/images/manuals/b5000/descrip/descrip.html`.

[Car96] Martin Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, June 1996.

[Cha95] Jeffrey Chase. *An Operating System Structure for Wide-Address Architectures*. PhD thesis, University of Washington, August 1995.

[Chu96] H. K. Jerry Chu. Zero-copy TCP in Solaris. In *USENIX Annual Technical Conference*, pages 253–264, 1996.

[Cif94] Christina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, July 1994.

[CKA91] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. *SIGPLAN Notices*, 26(4):224–234, 1991.

[CKD94] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–327, San Jose, California, 1994.

[DH66] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[EA03] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th Symposium on Operating Systems Design and Implementation*, 2003.

[ECC01] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP '01)*, pages 57–72, Oct. 2001.

[Fab74] Robert S. Fabry. Capability-based addressing. *CACM*, 17(7):403–412, July 1974.

[Fot61] John Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of backing store. *Communications of the ACM*, 4(10):435–436, October 1961.

[GN96] Dirk Grunwald and Richard Neves. Whole-program optimization for time and space efficient threads. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 50–59, 1996.

[HEV+98] 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.

[Hew02] Hewlett-Packard Corporation. *PA-RISC 2.0 architecture*, 2002.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schonberg, and Jean Wolter. The performance of microkernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Oct. 1997.

[HLH92] Erik Hagersten, Anders Landin, and Seif Haridi. DDM - a cache-only memory architecture. *IEEE Computer*, 25(9):44–54, 1992.

[HLP+00] Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-code performance is becoming important. In *Proceedings of the 1st Workshop on Industrial Experiences with Systems Software (WIESS)*, San Diego, CA, Oct. 2000. USENIX Association.

[HSH81] 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.

[IBM02] IBM Corporation. *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors G522-0290-01*, 2002.

[Int96] Intel. *Pentium Pro Family Developer's Manual, Vol 2*, 1996.

[Int97] Intel Corporation. Volume 1: Basic architecture. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 1997.

[Int02] Intel. *Intel Itanium Architecture Software Developer's Manual v2.1*, 2002.

[JJD+79] Anita K. Jones, Robert J. Chansler Jr., Ivor Durham, Karsten Schwans, and Seven Vegdahl. A multiprocessor operating system for the support of task forces. In *Proceedings of the 7th Symposium on Operating Systems Principles*, pages 117–127, Dec 1979.

[JKW95] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. *ACM Operating Systems Review, SIGOPS*, 29(5):213–226, 1995.

[JMG+02] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of USENIX annual technical conference*, June 2002.

[KCE92] 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.

[KH92] Gerry Kane and Joseph Heinrich. *MIPS RISC Architecture (R2000/R3000)*. Prentice Hall, 1992.

[Lam71] Butler Lampson. Protection. In *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, Princeton University, 1971.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

[Lev03] Markus Levy. ARM gets more deeply embedded. *Microprocesor Report*, 17(10):26–28, October 2003.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[Lie95] Jochen Liedtke. On micro-kernel construction. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 237–250, 1995.

[LLG+90] Daniel E. Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th annual international symposium on computer architecture (ISCA '90)*, pages 49–58, June 1990.

[MKF+98] Kenneth Mackenzie, John Kubiatowicz, Matthew Frank, Walter Lee, Victor Lee, Anant Agarwal, and M. Frans Kaashoek. Exploiting two-case delivery for fast protected messaging. In *HPCA*, pages 231–242, 1998.

[Moz03] Mozilla Organization. *Mozilla web browser*, 2003. `http://www.mozilla.org/`.

[MPC+02] Madanlal Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[MWA+96] James Montanaro, Richard T. Witek, Krishna Anne, Andrew J. Black, Elizabeth M. Cooper, Daniel W. Dobberpuhl, Paul M. Donahue, Jim Eno, Alejandro Fatell, Gregory W Hoeppner, Davidk Kruckmeyer, Thomas H. Lee, Peter Lin, Liam Madden, Daniel Murray, Mark Pearce, Sribalan Santhanam, Kathryn J. Snyder, Ray Stephany, and Stephen C. Thierauf. A 160MHz 32b 0.5W CMOS RISC Microprocessor. In *IEEE International Solid-State Circuits Conference, Slide Supplement*, February 1996.

[Nat97] National Software Testing Laboratories. *NSTL Final Report for Rational Software: Performance test of Rational Software's software product Purify*, October 1997. `http://www.rational.com/media/whitepapers/pnt-nstl.pdf`.

[Nec97] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, Jan 1997.

[Nel81] Bruce J. Nelson. *Remote procedure call.* PhD thesis, Carnegie-Mellon University, 1981. CMU-CS-81-119.

[ns00] NS Notes and Documentation. http://www.isi.edu/vint/nsnam/, 2000.

[PDZ00] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[PS03] Rina Panigrahy and Samar Sharma. Sorting and searching using ternary CAMs. *IEEE Micro*, 23(1):44–53, Jan/Feb 2003.

[RAB+03] Martin Rinard, C. Scott Ananian, Chandrasekhar Boyapati, Brian Demsky, Viktor Kuncak, Patrick Lam, Darko Marinov, Alex Salcianu, Karen Zee, and Wes Beebee. *The FLEX Compiler Infrastructure*, 1999–2003. `http://www.flex-compiler.csail.mit.edu`.

[Rat02] Rational Software Corporation. *Purify*, 2002. `http://www.rational.com/media/products/pqc/D610_PurifyPlus_unix.pdf`.

[Red74] David Redell. *Naming and Protection in Extendible Operating Systems*. PhD thesis, University of California, Berkeley, September 1974.

[REEBWG95] Mendel Rosenblum, Stephen A. Herrod Edourd E. Bugnion, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 285–298, Oct. 1995.

[RJ00] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.

[SA93] Rabin A. Sugumar and Santosh G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '93)*, pages 24–35, 1993.

[Sal74] Jerome H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[SBL03] Michael Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[SFL+94] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of architectural support for programming languages and operating systems (ASPLOS-VI)*, 1994.

[SGT96] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 174–185, 1996.

[Sha99] Jonathan S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999.

[SN02] Robert R. Schneck and George C. Necula. A gradual approach to a more trustworthy, yet scalable, proof-carrying code. In *Proceedings of Conference on Automated Deduction*, pages 47–62, 2002.

[Sof03] Software Engineering Institute. *CERT Coordination Center*, 2003. `http://www.cert.org/`.

[Sou03] Open Source. *bochs: The cross platform IA-32 emulator*, 2003. `http://bochs.sourceforge.net/`.

[SS75] 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. `http://web.mit.edu/Saltzer/www/publications/protection/index.html`.

[SSF99] 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.

[Sun96] Sun Microsystems. *UltraSPARC user's manual*, 1996.

[SW92] Walter R. Smith and Robert V. Welland. A model for address-oriented software and hardware. In *Proceedings of the 25th Hawaii International Conference on System Sciences (HICSS-25)*, volume 1, pages 720–729, January 1992.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[THK95] Madhusudhan Talluri, Mark D. Hill, and Yousef Y. A. Khalidi. A new page table for 64-bit address spaces. In *Symposium on Operating Systems Principles*, pages 184–200, 1995.

[Tor03] Linus Torvalds. *Linux kernel modules*, 2003. `http://www.kernel.org/`.

[vEBBV95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Symposium on Operating Systems Principles*, pages 303–316, 1995.

[vECC+99] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-kernel: A capability-based operating system for java. In *Secure Internet Programming*, pages 369–393, 1999.

[Wah92] Robert Wahbe. Efficient data breakpoints. In *Proceedings of the 5th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, Oct 1992.

[WCA02] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X)*, Oct 2002.

[WCC+74] William A. Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, July 1974.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *Proceedings of the 14th symposium on operating systems principles (SOSP)*, pages 203–216, December 1993.

[WN79] Maurice V. Wilkes and Roger M. Needham. *The Cambridge CAP Computer and Its Operating System*. North Holland, New York, 1979.

[WRBS00] Jonathan D. Pincus William R. Bush and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30(7):775–802, 2000.

[WSW+94] Robert Welland, Greg Seitz, Lieh-Wuu Wang, Landon Dyer, Tim Harrington, and Daniel Culbert. The newton operating system. In *Proceedings of the 39th IEEE Computer Society International Conference*, page ???, San Francisco, 1994.

[Zil01] Craig B. Zilles. Benchmark health considered harmful. *Computer Architecture News*, 29(3):4–5, 2001.