USENIX Association

Proceedings of
HotOS IX: The 9th Workshop on
Hot Topics in Operating Systems

Lihue, Hawaii, USA
May 18–21, 2003

Hardware Works, Software Doesn't: Enforcing Modularity with Mondriaan Memory Protection

Emmett Witchel and Krste Asanović
MIT Laboratory for Computer Science, Cambridge, MA 02139

Abstract

Two big problems with operating systems written in unsafe languages are that they crash too often and that adding features becomes much more difficult over time. One cause of both of these problems is the lack of enforceable memory protection between module boundaries. Clear module boundaries make dependencies explicit, resulting in more reliable and maintainable code. Mondriaan Memory Protection (MMP) is a hardware/software design for fine-grained memory protection that can enforce module boundaries for systems written in unsafe languages. We present the design of an MMP-based modular operating system kernel and show how MMP can be used to provide module isolation while maintaining performance.

1 Introduction

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 many users to lose data. The proliferation of embedded devices that manage non-transient data (like PDAs and digital cameras) translates lack of reliability into personal inconvenience. 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.

The largest problem for OS reliability is device drivers, which according to one study, can have three to seven times as many bugs as the rest of the kernel [3]. Many operating systems, like Linux, load drivers into the address space of the running kernel. This makes calling them efficient because they share the kernel address space and run with full kernel privileges. But it also makes them dangerous, as a single driver bug can crash the whole system. Drivers are often buggy because writing a correct driver requires knowledge of poorly documented features of the kernel programming environment, and drivers are often written by device manufacturers who are not seasoned kernel developers.

Mondriaan Memory Protection (MMP) is a fine-grained hardware memory protection scheme that equips OS developers with a powerful and simple tool to increase reliability [7]. From an OS developer's perspective, MMP supersedes the protection part of a page table, providing permissions granularity down to single 32-bit words. It does not replace the page table structure, which is still needed if the system requires virtual address translation.

In this paper, we describe how MMP can be used to increase the robustness of an operating system, without compromising its performance. We can enforce the existing boundaries between dynamically loaded kernel modules and the core of the kernel, which is currently only protected by programmer convention. Although memory corruption is only one possible failure mode of a poorly behaved device driver (others include leaving interrupts disabled, breaking the lock discipline, or excessive resource consumption), it is the most common and the most difficult to guard against efficiently. Once boundaries with kernel modules are enforceable, we can begin dividing the core kernel into protected subcomponents to improve maintainability.

2 Mondriaan Memory Protection

MMP is an efficient, fine-grained memory protection system that allows word-level protection at word boundaries. The MMP architecture was described in detail in [7], but here we give a brief review.

Figure 1 shows the architectural structure of an MMP system. A *protection domain* (PD) contains a map from address to permissions for the entire address space. The map is stored in a *permissions table* held in kernel memory. The table is similar to the permissions part of a page table, but permissions are kept for individual words (32-bits) in an MMP system. A CPU control register holds the base address of the active domain's permissions table. MMP can be used in systems with a single virtual or physical address space, or systems with multiple virtual address spaces in which case a PD lives within one address space.

The CPU contains a hardware control register which

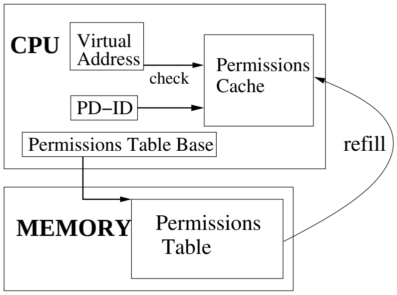

Figure 1: The major components of the Mondriaan memory protection system. On a memory reference or instruction execution, the processor checks a hardware permissions cache. If the permissions information is not in the cache, it is fetched from the table in memory.

holds the protection domain ID (PD-ID) within which the currently running thread executes [5]. The MMP hardware on the CPU checks every memory access to see if the active domain has appropriate permissions for that access. Every instruction fetch is also checked for execute permissions.

The processor maintains a hardware permissions cache, the *protection lookaside buffer* (PLB) [5], to accelerate permission checks. Detailed simulations show this cache to be effective, resulting in only an 8% increase in memory traffic due to PLB misses even when fine-grained permissions are used extensively [7]. Permissions checks only need to be completed before instruction commit and so are not on the critical path of the processor. We expect a modern out-of-order processor would hide some or all of the latency of the references to the permissions table, reducing the performance impact of the additional memory references.

Frequent permissions changes might require hardware support for PLB consistency in multiprocessors. The form of this support is still an open question.

MMP is backwards compatible with current instruction sets and binaries. CPUs could be made with TLB permissions and MMP permissions to allow full binary compatibility. Or MMP could replace TLB permissions with support from the device dependent layer of the kernel, and a few trusted applications like the system loader and the dynamic linker.

We preserve 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) can be 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 and is special in that it can access all of memory without the mediation of a permissions table.

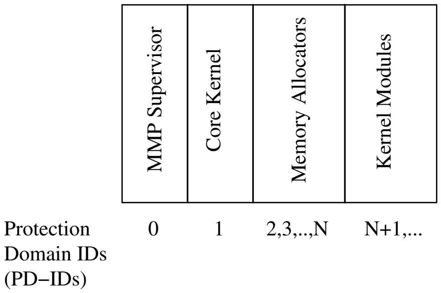

Figure 2: How the kernel address space can be divided into protection domains. The memory supervisor is in protection domain 0, and has unfettered access to memory. The bulk of the kernel is loaded into the first created protection domain (PD-ID 1). Then it loads other pieces of itself, like the memory allocators, into other domains. Finally kernel modules are loaded on demand.

3 Using MMP in the Kernel

In this section, we describe the design of an MMP-based operating system kernel.

3.1 Memory Supervisor

Figure 2 shows how the kernel address space can be split into multiple protection domains. The MMP memory supervisor domain (PD-ID 0) manages the memory permissions tables and provides an API to control memory permissions. In this section, we describe the MMP memory supervisor API by showing how it would be used during the boot of a modularized kernel.

At system reset, the processor starts running at the reset vector in PD-ID 0. The BIOS loads 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.

The MMP hardware checks all processor memory accesses against the values stored in the current permissions table (except those made by domain 0). The MMP supervisor software can enforce additional memory usage policies because all calls for permissions manipulation are made via the supervisor. The supervisor will reject requests that violate its policy. Just because the supervisor exports an API does not mean that all created domains have permissions to call into it. As we will see, it is possible to construct a domain which does not have permission to call into the supervisor, forcing memory management to happen via the intermediary of the creating domain.

For example, the supervisor tracks *ownership* of memory regions. A domain obtains ownership after allocating a new memory region from the supervisor, or when another domain grants ownership of a region. Only the owner of a memory region may revoke permissions, or grant ownership to another domain.

Once initialized, the supervisor creates a new domain (PD-ID=1) to hold code and data for the core of the kernel. Protection domain creation is provided by the `mmp_alloc_PD(user/kernel)` supervisor call, which returns a PD-ID. The supervisor does not allow a user domain to create a kernel domain.

To start the kernel, the supervisor must first load the boot loader into PD 1. Initially, a PD has no permissions to access memory. In order for the boot loader to run, it will need execute permission on its code, read and write permissions on its data, a read-write stack, and possibly a read-write heap. Setting permissions is done by the `mmp_set_perm(ptr, length, perm, PD-ID)` routine. The memory supervisor uses the `mmp_set_perm` call to establish proper permissions for boot loader execution. The supervisor then performs a cross-domain call (described below) to transfer control to the boot loader which now runs in a protected kernel domain (PD-ID=1).

The boot loader wants to load the core kernel, and so needs to ask the memory supervisor for additional memory space. The `mmp_alloc(n_bytes)` supervisor function allocates a region of memory and returns a pointer (a variant of `mmp_alloc` allows a desired address placement to be supplied). The supervisor records PD-ID 1 as the owner of this memory region. The owner of a region can call `mmp_free(ptr)` to release a memory region back to the supervisor (`mmp_free` can also take an optional length parameter, allowing partial resources to be reclaimed).

Once the core kernel starts running, it can create child PDs to hold kernel modules. The core kernel will want to export permissions for portions of its address space to its child modules using the `mmp_set_perm` call, and it might also want to pass on ownership of memory regions to kernel modules, to allow them to manage the permissions of their children. The `mmp_mem_chown(ptr, length, PD-ID)` call passes ownership of a memory region to the protection domain given by the PD-ID parameter. Although a kernel module could be allowed to ask the supervisor for memory regions directly, usually the core kernel will manage memory usage of its modules. The core kernel can block kernel modules from calling the memory supervisor by not exporting call permission on the supervisor entry points to the kernel modules.

The `mmp_set_perm` call supports a `transitive` flag which, in addition to exporting permissions, also allows the receiving domain to transitively export permissions. This allows calling domains to either enforce a policy of only allowing a particular service domain (perhaps one containing cryptographically verified code) to implement a function, or allowing a service domain to subcontract work to other service domains. Transitive permissions are still distinct from ownership because only the owner can return memory to domain 0, and a domain that receives transitive permissions can not revoke permissions from a domain higher on the receiving chain.

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(recursive)` on their children. If the recursive flag is true, all of the deleted protection domain's children are also deleted. Otherwise, they are reparented to the closest surviving parent remaining in the tree.

One important special case for sharing data is global read-only access. MMP supports export of data to all protection domains read-only. When a piece of memory is exported globally, the supervisor adds the permissions to all existing permissions tables. It also tracks the global export in a table so it can add permissions for this globally exported memory to new protection domains as they are created.

3.2 Stacks and threads

Code and heap data regions can be associated with a protection domain, and are typically owned by one domain and exported to others. Stack storage must be managed differently, however, because stacks are used by threads that move between protection domains. In our MMP OS design, stack storage is owned by the memory supervisor. To get stack storage, a thread manager in a protection domain calls `mmp_alloc(stack)`. The `stack` flag tells the supervisor that this is a stack segment, a fact which the supervisor records while maintaining ownership of the storage.

The memory supervisor only owns and manages the stack space for each thread. Other details about the thread, like its control block and the scheduling policy that govern it, are determined by the kernel or an arbitrary thread-managing domain.

Stack permissions are managed by the supervisor call `mmp_supr_set_perm(ptr, length, perm, exclusive, PD-ID)`, which is like the `mmp_set_perm` call, but the `mmp_set_perm` call only works for memory that is owned by the caller. The `mmp_supr_set_perm` call requests that the supervisor make a permissions change on memory that it owns. Of course, the supervisor range checks the address and refuses action if the request is inappropriate. The `exclusive` flag requests that all permissions for other PDs be revoked.

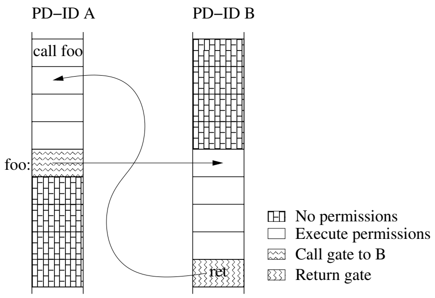

Figure 3: How MMP is used for cross-domain calling. The arrows indicate a domain crossing, which is handled by hardware. The call gate has the destination PD-ID stored in a special record in the MMP protection table. The return gate verifies that it is returning to the last call on the cross-domain call stack (not pictured).

When a stack segment is allocated, the supervisor records the creating protection domain, and a stack-ID, which is just the base address of the stack segment. 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 has permissions to make this stack segment active. When the supervisor receives a call to set stack permissions, it checks that the request is for the active stack.

Although this scheme prevents one thread changing permissions on another thread's stack, when multiple threads run in the same protection domain they can still potentially access each other's stack frames. This is a minor protection violation, as the code in the domain has already been trusted with the stack frames. But we can eliminate this violation by adding another hardware mechanism, discussed in Section 3.3.

3.3 Cross-domain calls

Threads move between protection domains by performing a cross-domain call as shown in Figure 3. The cross domain call can be initiated by any control flow instruction, though it will usually be a standard subroutine call instruction. The target of the call is marked with a special permissions value known as a *call gate*. A call gate also has the PD-ID of the target protection domain stored in a special record type in the permissions table.

When call gate permissions are detected on a subroutine call, the hardware atomically pushes the return address and the caller's PD-ID onto a stack that resides in protected storage. This stack is called the *cross-domain call stack* and is implemented with some combination of an on-chip top-of-stack buffer, backed up by off-chip protected memory. The architecture then reads the new (callee's) PD-ID value from the permissions table and copies this into the CPU's PD-ID register. It looks up the protection table base pointer for the new PD-ID, and stores it in the table base register. At the end of this process, instructions are fetched from the context of the new protection domain.

MMP also uses return gates, which are the duals of call gates. They are also implemented using standard instructions and special MMP protection values. A return gate causes the architecture to pop the cross-domain call stack, finding the return address and protection domain of the last call. The architecture checks the return address against the return address being used by the return instruction. If they are different, a fault is generated which is handled by the memory supervisor. If the return addresses match, the hardware sets the protection domain to the PD-ID that was popped off the stack.

Call and return gates provide an efficient mechanism for mutually distrustful protection domains to safely call each other's services, without requiring new instructions in the ISA. Cross-domain calls are analogous to lightweight remote procedure calls, though cross-domain calls do not require copying data for protection, or an argument stack per domain pair, as LRPC does.

We expect cross-domain calls to be fast because the amount of on-chip state that needs to be changed is small. We believe CPU designers will be motivated to accelerate cross-domain calls to enable the benefits of protected execution. For example, traditional CPU microarchitectures flush pipelines on a context switch, imposing a large overhead. Domain switches can be made considerably faster by associating PD-ID values with each instruction in the pipeline, reducing the need to flush the pipeline.

If the called function needs an activation frame, it must request permissions for the stack space, and also make sure that permissions for the frame are exclusive to the current thread. This is done using the `exclusive` flag in a call to `mmp_supr_set_perm`. Because domains take exclusive access to a frame before executing in the frame, a frame's permissions do not need to be revoked at the end of a function for the caller's safety. A callee that is concerned about security could overwrite its activation frame before returning to avoid leaking information.

Calls to establish a frame will be frequent and could potentially be expensive. Two special hardware registers can make the creation of a frame fast, and can make permissions to read and write the frame thread-local, closing the security loophole discussed in Section 3.2.

When the supervisor makes a stack current for a given CPU, it fills in two registers—frame base **fb**, and stack limit **sl**. The hardware allows read 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. Its initial value for a given thread's quantum is specified (as a parameter to mmp_set_stack) when the thread manager starts the thread. The memory supervisor verifies the initial value of **fb** to make sure it is within the stack segment that is being activated. On a cross-domain call, the current **fb** is pushed onto the cross-domain call stack, and the current stack pointer is made the new **fb**. The hardware checks that the new **fb** value is smaller than the old value. Thus the hardware insures that the stack grows down, and the memory supervisor insures that it starts and ends in the right place, so the two registers can only be used to gain permission to read and write stack memory. The registers become part of the thread state which must be saved and restored.

3.3.1 Passing arguments

Heap data is owned by a protection domain, so cross-domain sharing of heap data is straightforward—the caller exports permissions to the callee. This is a lightweight form of argument marshaling that does not require data copying or even data structure traversal (for many data structures). Domains can set up shared buffers in advance of the cross-domain call. 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.

Passing arguments on the stack is more complicated because a protection domain does not own the stack. To pass arguments on the stack, a cross-domain call must be preceded by a call to the supervisor to export any permissions that might be required for the call to work properly, e.g., giving read-write permissions on a stack structure whose address is being passed as a parameter.

Calls to export permissions on a stack frame are stylized and so can be highly optimized. For instance many custom entry points could be provided which take the current top of the stack as the only parameter. They would establish read-only permission for a fixed small number of stack slots by writing directly into the permissions tables. Most calls would fit into one of these patterns, but for those that did not, the dynamic linker could request the generation of a custom entry point for a given stack layout.

We can reduce the number of exports for inter-module calls by hoisting the export out of a loop, reusing stack slots for different inter-module calls, and not changing the permissions until right before the call if the call is unlikely to execute.

3.4 Space overhead of protection domains

For its finest-grained permissions tables, MMP stores two bits of permissions data per 32-bit word, so the space cost for the tables is $\frac{1}{16}$, or 6.25%. Applications that use coarse-grained permissions regions can experience less than 6.25% space overhead, potentially much less (e.g less than 0.7% for putting each program section in its own protection region [7]). The space overhead of the tables is proportional to how densely the address space is being used, with lower density leading to higher overhead (just as with page tables).

If multiple protection domains are arranged densely in the address space (as kernel module code and data are arranged in Linux), then there is little additional space cost to dividing domains. Each new domain requires a root table, which has a fixed cost of 4KB (though they can be made smaller if need be). The root tables need to be stored in unmapped kernel memory, but user root tables can be swapped. Domains that share a permissions view for much of memory can share permissions tables below the root level.

4 Adding MMP to Linux

We split the Debian Linux kernel version 2.4.19 into different domains, putting the core kernel in one domain and placing each loaded kernel module in its own, separate, domain. Code and data exports were derived from tools that interpreted the symbol information in the kernel modules. Because so little code and data is actually imported or exported by any module (relative to what is available in the kernel address space), restricting access to those symbols results in a large gain in modularity.

For instance, most modules import the kernel function printk so they can log errors. We treat the unresolved symbol in the module as a request for permissions to call the routine. While this works well for code symbols, data boundaries are less likely to be completely characterized by symbol information because a module might dereference a pointer from an imported structure, reading memory outside that structure.

We booted the OS on bochs, a complete machine simulator, and measured domain crossings. Our rough prototype implemented all of MMP in the hardware model (including table management, which really belongs in OS code). The OS boot from disk shares many properties with any disk intensive workload. There were 284,822 protection domain changes in the boot, 97.5% of which were to or from the IDE driver. About 1 billion instructions were executed (955,240,000), yielding an average of 3,353 instructions executed in each domain. This demonstrates a surprisingly fine-grained interleaving of module execution and underscores the need for efficient cross-domain calling, justifying hardware support.

MMP not only enforces memory safety, it enables performance optimizations. For instance, one reason the kernel needs a copyin procedure is because it can not trust the user to put their data in the right spot and not corrupt kernel data structures. With MMP, we can change the interface to allow the user to write their data into kernel space directly, and still protect kernel data structures.

5 Comparison with other protection mechanisms

Nooks [6] provides device driver safety using conventional hardware. MMP can reduce the sometimes substantial performance overheads Nooks endures to run on current hardware. Also, MMP has many uses in addition to device driver safety.

Palladium [2] used x86 hardware protection for intermodule protection both in the kernel and at user level. It is very difficult to use x86 segmentation to implement protected shared libraries (Palladium used page-based protection for shared libraries), and using it in the kernel complicates the programming model for extensions.

Capabilities [4] are a fine-grained protection mechanism that OS designers have used to build big systems (e.g., IBM's AS400). Capabilities are special pointers that contains both location and protection information for a segment. Capabilities have known disadvantages such as difficulty with rights revocation, requiring tagged memory, and difficulty for two domains to share a data structure (with embedded capabilities) with different permissions.

Lightweight remote procedure call (LRPC) [1] 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.

There are a variety of safe language approaches for OS extensibility and all of these approaches have common problems—excessive CPU and memory consumption is common in safe languages or unsafe languages retrofited with type information. A safe language restricts an implementation to a single language, it ignores a large base of existent 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.

A deeper problem with language-only safety is the size of the system that must be trusted. For an MMP system, one must trust the MMP hardware and the MMP supervisor software. These are likely to be much simpler and more amenable to verification than a language compiler and runtime. This is especially true for optimized safe-language implementations which employ complex analyses to improve runtime efficiency.

Modern static analysis and model checking tools can scale sufficiently to deal with large OS codes. These systems can find many important bugs without flooding the user with false positives. But they do suffer from false negatives, and are therefore compatible with and benefit from the dynamic checking of an MMP system.

6 Conclusion

MMP allows systems to be extensible, efficient, and robust, whereas current software-based schemes require that a designer choose only two of these properties. Compared with other proposed hardware fine-grained protection schemes, MMP has the advantage that it is backwards compatible with existing instruction sets and existing OS protection models, and so can be introduced incrementally.

MMP enables the hardware to enforce the intermodule boundaries already present in the software structure, helping to address the problem of poor OS stability due to poorly coded device drivers. MMP supports further modularization of the kernel by reducing the overhead of enforced modularity, which should result in systems that are more reliable and more maintainable.

References

[1] E.D. Lazowska B.N. Bershad, T.E. Anderson and H.M. Levy. Lightweight remote procedure call. In *SOSP-12*, 1989.

[2] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *SOSP-17*, 1999.

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems error. In *SOSP-18*, 2001.

[4] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9(3):143–155, March 1966.

[5] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architectural support for single address space operating systems. In *ASPLOS-V*, 1992.

[6] M. Swift, S. Martin, H. M. Levy, and Susan J. Eggers. Nooks: an architecture for reliable device drivers. In *Proceedings SIGOPS-10*, 2002.

[7] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *ASPLOS-X*, Oct 2002.