Sentry: Light-Weight Auxiliary Memory Access Control*

Arrvindh Shriraman
Department of Computer Science
University of Rochester
ashriram@cs.rochester.edu

Sandhya Dwarkadas
Department of Computer Science
University of Rochester
sandhya@cs.rochester.edu

ABSTRACT

Light-weight, flexible access control, which allows software to regulate reads and writes to any granularity of memory region, can help improve the reliability of today's multi-module multi-programmer applications, as well as the efficiency of software debugging tools. Unfortunately, access control in today's processors is tied to support for virtual memory, making its use both heavy weight and coarse grain. In this paper, we propose Sentry, an auxiliary level of virtual memory tagging that is entirely subordinate to existing virtual memory-based protection mechanisms and can be manipulated at the user level. We implement these tags in a complexity-effective manner using an M-cache (metadata cache) structure that only intervenes on L1 misses, thereby minimizing changes to the processor core. Existing cache coherence states are repurposed to implicitly validate permissions for L1 hits. Sentry achieves its goal of flexible and light-weight access control without disrupting existing inter-application protection, sidestepping the challenges associated with adding a new protection framework to an existing operating system.

We illustrate the benefits of our design point using 1) an Apache-based web server that uses the M-cache to enforce protection boundaries among its modules and 2) a watchpoint-based tool to demonstrate low-overhead debugging. Protection is achieved with very few changes to the source code, no changes to the programming model, minimal modifications to the operating system, and with low overhead incurred only when accessing memory regions for which the additional level of access control is enabled.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design Styles—Shared memory; C.0 [General]: Hardware/Software interfaces; C.1.2 [Processor Architectures]: Multiprocessors; D.2.0 [Software Engineering]: General — Protection mechanisms

General Terms: Performance, Design, Reliability, Security

Keywords: Access control, Cache coherence, Multiprocessors, Memory protection, Sentry, Protection domains, Safety.

*This work was supported in part by NSF grants CCF-0702505, CNS-0411127, CNS-0615139, CNS-0834451, and CNS-0509270; NIH grants 5 R21 GM079259-02 and 1 R21 HG004648-01; and equipment support from Sun Microsystems Laboratories.

ISCA'10, June 19–23, 2010, Saint-Malo, France.

1. INTRODUCTION

Modern software are complex artifacts consisting of millions of lines of code written by many developers. Developing correct, high performance, and reliable code has thus become increasingly challenging. The prevalence of multi-core processors has increased the burden on software developers. Fine-grain intra- and inter-thread interactions via memory make it difficult for developers to track, debug, and regulate the accesses arising from the various software modules. Architectural features for *access control* that enable software to track and regulate accesses will help develop more robust and fault tolerant applications.

As one example, Figure 1 presents a high-level representation of the developers' view of Apache [31]. The system designers specify a software interface that defines the set of functions and data that are private and/or exported to other modules. For the sake of programming simplicity and performance, current implementations of Apache run all modules in a single process and rely on adherence to the module interface to protect modules. Unfortunately, a bug or safety problem in any module could potentially (and does) violate the interface and affect the whole application.

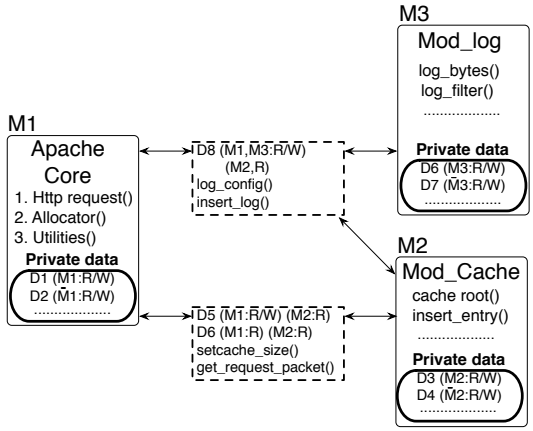


Figure 1: Example of software modules in Apache. M1,M2,M3 — modules, D1...D8 — data elements. Dashed lines indicate data and functions shared between modules. Tuple D: (M:P) indicates module M has permission P on the memory location D. R - Read only ; R/W - Read or Write

There are two key aspects in regulating the accesses to a specific datum: (1) the domain, referring to the environment or module, that a thread performing the access is executing in; and (2) the access privileges, the set of permissions available to a domain for the object [10]. Access control is used to enforce the permissions specified for the domain and object. An instance of the use of flexible access control is enforcing the rule that a plug-in should not have access to the application's data or another plug-in's data, while the application could have unrestricted access to all data. Access control can also be used for program debugging in order to efficiently

intercept accesses and detect violations of specific invariants, such as accesses to uninitialized or unallocated locations, dangling references, unexpected write values, etc. Tracking memory bugs is an intensive task and it is especially complicated in module-based applications where memory is accessed at many different sites.

Access control mechanisms in current general-purpose processors consist of either operating system (OS) page protection bits or a limited number of word-granularity watchpoint registers (4 on the x86). The former is restricted to providing page-granularity protection while the latter's usefulness is handicapped by the small number of watchpoint registers. Unfortunately, the protection targets in most realistic applications are often more fine-grain, at the granularity of a few words. Also another consideration in protection usage is "thread-awareness", which would need to allow different application threads to be regulated differently for the same location and also allow the same thread to have different access permissions to different locations. Finally, hardcoding the number of protection levels and semantics has lead to restricted use of the mechanism. Overall, existing systems do not satisfy modular software's demands for flexible memory protection.

Currently, flexible access control typically relies on software instrumentation tools, which can cause significant slowdown (e.g., 22—60$\times$ [14, 27]). Recent work [6] has reduced the overheads by focusing on the integrity of data (no unsafe writes) and compromising the safety (permit unsafe reads). Software-based access control also need synchronization to ensure the atomicity of metadata operations and data access in multi-threaded workloads. Recent hardware proposals for fine-grain access control such as employed by Mondrian [24] seek to provide a flexible protection framework. However, such a design also introduces additional permission tag structures within the highly optimized processor pipeline. The permission tags are accessed on every memory operation, which results in extra work and energy consumption. Finally, they add new layers of protection to replace OS processes and require every application, regardless of its desire for flexible access control, to use the new interface.

In this paper, we investigate *Sentry*, a hardware framework that enables software to enforce protection policies at runtime. The core developer annotates the program to define the policy and then the system ensures the privacy and integrity of a module's private data (no external reads or writes), the safety of inter-module shared data (by enforcing permissions specified by the application), and adherence to the module's interface (controlled function call points).

We propose a light-weight, multi-purpose access control mechanism that is independent of and subordinate to (enforced after) the page protection in the Translation-Lookaside-Buffer (TLB). We implement Sentry using a permissions metadata cache (*M-cache*) that intercepts only L1 misses and resides entirely outside the processor core. It reuses the L1 cache coherence states in a novel manner to enforce permissions and elide checks on L1 hits. Permission metadata checks thus require between 1 and 10% (depending on workload) of the energy required by the in-processor option. Since the M-cache is on the L1-miss path it places fewer constraints on the pipeline cycle compared to per-access in-processor checks [24]. Finally, the M-cache also exploits cache coherence to implicitly invalidate cached metadata on permission changes (no need for heavyweight software shootdowns).

From the software's perspective, Sentry is a pluggable access control mechanism for application-level fine-grain protection. It works as a supplement (not a replacement) to OS process-based protection and this leads to three main advantages. First, Sentry incurs space and performance overhead only when additional fine-grain protection is needed as otherwise existing page-based access control can be used. Second, the software runtime that manages the intra-application protection can reside entirely at the user level and can operate without much OS intervention, making the system both efficient and attractive to mainstream OS vendors. Third, within the same system, applications using the Sentry protection framework can co-exist with applications that do not use Sentry's services. We demonstrate that Sentry can realize various intra-application protection models with simple permissions annotations and does not require any changes to the programming model or language.

We used Sentry to enforce a protection model for the Apache web server. We primarily safeguard the core web server's data from extension modules by ensuring that modules do not violate the specified interface. We achieved this without requiring any changes to the programming model, with minimal source code annotations, and moderate performance overheads ($\simeq$13%). We note that our efforts to compare against process-based protection (i.e., enclose the module in a separate process and use RPC for web server to module communication) were frustrated by the programming model and cumbersome non-shared-memory interface. We also validate the suitability of Sentry when employed for a watchpoint-based memory debugger. We estimate that its overheads are comparable to proposed debugging hardware (6%) and are much lower than proposed fine-grain protection hardware (half the overhead) or software approaches.

Section 2 presents the design of the M-cache hardware and describes the M-cache operations. Section 3 describes the various Sentry protection models and the support needed from software. Section 4 demonstrates the applications and evaluates Sentry's performance overheads. Section 5 discusses related work and we conclude in Section 6.

2. SENTRY: AUXILIARY MEMORY ACCESS CONTROL

Access control must essentially provide a way for software to express permissions to a given location for the active threads. Hardware is expected to intercept the accesses, check them against the expressed permissions, and raise an exception if the thread does not have the appropriate permissions. Current processors typically implement a physical memory hierarchy through which an access traverses looking for data. Accesses can be intercepted at any level in the memory hierarchy: within the processor core, in the L1 miss path (this paper), or any other level.

Why not within the processor?

Most protection schemes (e.g., Mondrian [24], Loki [26], TLB) adopt the logically simple option of intercepting all memory accesses within the processor. However, this requires additional structures in highly optimized stages of the processor pipeline. Since the access control hardware is consulted in parallel with L1 access on every load and store, the hardware design is constrained by the need to stay within the 1-2 cycle L1 access latency. High-performance transistor technology will likely be employed, leading to a noticeable energy and area budget.

Access Control on L1 misses

In Sentry, we have chosen to place the access checks on the L1 miss path to avoid processor core modifications. The L1 miss rate in most applications is a small percentage of the total accesses (1-4% for PARSEC, 4-9% for SPLASH2, 16% for Apache). Since the checks occur only on L1 misses and in parallel with L2 accesses (which can take 30—40 cycles), we have a longer time window to complete the checks and can trade performance for energy. Employing energy-optimized transistor technology [13] can save leak-

age and dynamic power. We found that all things equal (granularity of protection, number of entries), the L1 miss path option consumes only 1—10% of the power of an approach implementing the checks within the processor. Placing the access control hardware any further down (say on the L2 miss path) complicates the processor interface (for example, when propagating exceptions back to software). Furthermore, protection changes which require invalidation of in-cache data causes higher overhead, since an eviction at the higher cache level incurs higher penalty.

Our design choice to intercept L1 misses implies that the smallest protection granularity we support is an L1 cache line. Sub-cache-line granularity can be supported at the cost of either additional bits in the L1 or software disambiguation. To accommodate L1 cache line granularity monitoring (typically 16 - 64 bytes), the memory allocator can be modified to create cache-line-aligned regions with the help of software annotations (compiler or programmer). Software can also further disambiguate an access exception in order to implement word granularity monitoring if necessary.

2.1 Metadata Hardware Cache (M-Cache)

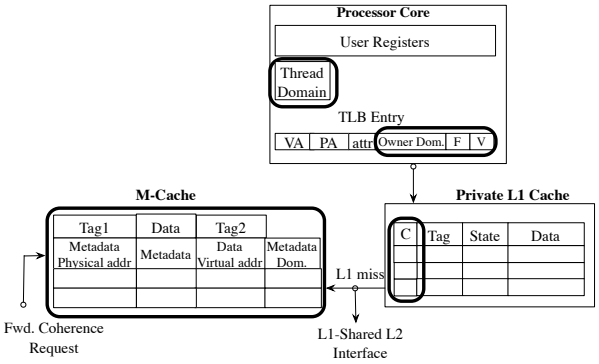


Figure 2: M-cache Mechanism. Dark lines enclose add-ons. Dom.: Domain.

In Sentry, we maintain the invariant that a thread is allowed to access the L1 cache without requiring any checks i.e., L1 accesses are implicitly validated. To implement this we include a new hardware metadata cache (M-cache) that controls whether data is allowed into the L1.

Figure 2 shows the M-cache positioned in the L1 miss path outside the processor core. Each entry in the M-cache provides permission metadata for a specific virtual page. The per-page metadata varies based on the intended protection model (see Section 3). In general, it is a bitmap data structure that encodes 2 bits of access permission (read-only, read/write, no access, execute) per cache line. Note that these are auxiliary to the OS protection scheme since the access has to pass the TLB checks before getting to the memory system. This ensures that the M-cache's policies don't interfere with the OS protection scheme. An M-cache entry is tagged on the processor side using the virtual address (VA) of the page requiring the permission check. Using virtual addresses allows user-level examination and manipulation of the M-cache entries and avoids the need to purge the M-cache on a page swap. The M-cache also contains a network-side tag consisting of the physical address of the metadata (MPA) to ensure entries are invalidated if the metadata is changed (more details in Section 2.4). The domain id fields included in the processor register, TLB entry, and M-cache, implement the concept of protection domains (we discuss this in more detail in Section 3). Table 1 lists the ISA interface of the M-cache.

Dual tags

The dual tagging of the M-cache does introduce an interesting challenge. We need to ensure that when the entry in one of the tag arrays is invalidated (VA due to processor action or MPA due to network message), the corresponding entry in the other tag array is also cleaned up. One option is to use a fully associative MPA tag array, but this is expensive in terms of both energy and area. Assuming limited associativity, the VA, the data address using the access control metadata, and the MPA, the metadata word's physical address, may index into different positions in their respective tag arrays. To solve this issue, we include forward and back pointers similar to those proposed by Goodman [7] — a VA tag (with which the metadata is associated) includes a pointer (set index and way position) to the corresponding entry in the MPA array and the MPA tag entry includes a back pointer. The dual tagging could result in additional conflict evictions due to collisions in the VA and MPA mappings.

2.2 When is the access checked?

Sentry creates permissions metadata only when an application requires fine-grain or user-level access control. To achieve this, we use an unused bit in the page-table entry to encode an "F" bit (Fine-grain). If the software doesn't need the checks, it leaves the 'F" bit unset and hardware bypasses the M-cache for accesses to the page. [1] If the bit is set, on an L1 miss, the M-cache is indexed using the virtual address of the access to examine the metadata. The virtual address needed by the M-cache lookup is available at the load-store unit for the L1 miss. Once a cache block is loaded into the processor's L1 cache, we use coherence's MESI states to transparently enforce permissions (i.e., cache hits don't check the M-cache). Without involving any modifications to the coherence protocol, M or E can support read/write permissions, S can enforce read-only, and I can check any permission type. An attempt to access data without the appropriate coherence state will result in an L1 miss, which checks the M-cache.

Data persistence in caches does introduce a challenge; data fetched into the L1 by one thread continues to be visible to other threads on the processor. Essentially, if two threads have different access permissions, the L1 hits could circumvent the access control mechanism. Consider two threads T1 and T2, which the application has dictated to have read-write and read-only rights to location A. T1 runs on a processor, stores A, and caches it in the "M" state. Subsequently, the OS switches in T2 to the same processor. Now, if T2 was allowed to write A, the permission mechanism would be circumvented. To ensure complexity-effectiveness, we need to guard against this without requiring that all L1 hits check the M-cache. We employ two bits, a "V" (Verify) bit in the page-table to indicate whether the page has different permissions for different threads and "C" (checked) bit in the cache tag, which indicates if the cache block has already been verified (1 yes, 0 no). All accesses (hit or miss) check permissions if the TLB entry's "V" bit is set and the "C" bit in the L1 cache tag is unset, indicating first access within the thread context. Once the first access verifies permissions, the "C" bit is set. This ensures that subsequent L1 hits to the cache line need not access the M-cache. The "C" bit of all cached lines is flash-cleared on context switches.

Exception Trigger

When an access does not have appropriate rights, the hardware triggers a permission exception, which occurs logically before the access. These exceptions are entirely at the user level and can use

[1] We also use this technique to prevent accesses on the permissions metadata from having to look for metadata.

Registers	
%mcache_handlerPC:	address of handler to be called on a user-space alert
%Domain1_Handler:	address of handler to be called within user-level supervisor (see Section 3.3)
%mcache_faultPC:	PC of faulting instruction
%mcache_faultAddress:	virtual address that experienced the access exception
%mcache_faultInstructionType:	read, write, or execute
%mcache_faultType:	M-cache miss or permission exception
%mcache_entry:	per-page permissions metadata; 2 bits / cache line to represent Read-only, Read/Write, Execute, and No access)
%mcache_index	index into the M-cache
Instructions	
get_entry vaddr,%mcache_index	get an entry for vaddr and store its index in %mcache_index
inv_entry vaddr,%mcache_index	evict vaddr's metadata from M-cache and return its position in %mcache_index
LD vaddr, %mcache_index	load vaddr into M-cache position pointed to by %mcache_index
LL vaddr, %mcache_index	Load Linked version of the above
LD_MPA vaddr,%mcache_index	load physical address of cache block corresponding to vaddr into MPA
ST %mcache_index,%mcache_entry	store the data in %mcache_entry into the M-cache entry pointed to by %mcache_index
SC %mcache_index,%mcache_entry	Store-Conditional version of the above
switch_call %R1, %R2	Effects a subroutine call to the address in %R1 and changes thread domain to that specified in %R2 (see Section 3.3)

Table 1: M-cache Interface. The %Domain1_Handler register and switch_call instruction are discussed in detail in Section 3.3.

the same stack, privilege level, and address space as the thread making the access. Our design reuses the exception mechanism on modern processors. The M-cache response marks the instruction as an exception point in the reorder buffer. The permissions checks are enforced at instruction retirement, at which time the exception type is checked and a software handler is triggered, if needed. The permission check, which is performed by looking up either the L1 cache state on a hit or the M-cache metadata on a miss, has potential impact only at the back end of the pipeline in the case of a miss. Pipeline critical paths and thereby cycle time consequently remain unaffected. Note that permission violations can be detected at the execution stage and the line will be brought into the appropriate state if it doesn't have permissions. (e.g., read-only permissions means line is in "S" state, no permissions means the line is not allocated).

On a permission exception, the M-cache provides the following information in registers: the program counter of the instruction that failed the check (%mcache_faultPC) , the address accessed (%mcache_faultAddress), and the type of instruction (%mcache_faultInstructionType). There are separate sets of registers for kernel and user-mode exceptions.

2.3 How is the M-cache filled?

The M-cache entries can be indexed and written under software control similar to a software TLB. Allowing software to directly write M-cache entries (1) allows maintenance of the metadata in virtual memory regions using any data structure and (2) permits flexibility in setting permission policies. The former decouples the metadata size from the limitations of the hardware M-cache size, allowing the protection metadata to grow arbitrarily under software control and to be structured to suit the specific applications' need. Hardware never updates the M-cache entries and it is expected that software already has a consistent copy. Hence, any evictions from the M-cache are silently dropped (no writebacks). Since no controller is required to handle refills or eviction, the implementation is simplified.

The ISA extensions (see Table 1) required are similar to the software-loaded TLB found across many RISC architectures (e.g., SPARC, Alpha, MIPS). There are separate load and store instructions that access the metadata using an index into the M-cache. In addition, to fill the MPA (not typically found in the TLB) we use an instruction (LD_MPA) that specifies the virtual address of the physical address for which a tag needs to be set up in the MPA.[2]

[2]Hardware also sets up pointer to the virtual address tags so that invalidations can cleanup entries consistently in both tag arrays (see Section 2.1).

M-Cache Fill Routine()
/*X: Virtual address of data */

```
1. get_entry X,%mcache_index
2. LL X,%mcache_index
3. P = get_permissions(X)
4. LD P,%mcache_entry
5. LD_MPA P,%mcache_index
6. SC %mcache_index,%mcache_entry
7. if failed(SC) goto 1;
```

Figure 3: Pseudocode for inserting a new M-cache entry.

The MPA is used by the hardware to ensure the entry is invalidated if the metadata changes, i.e., when a store occurs anywhere in the system to the MPA address. Typically, the metadata is maintained in the virtual address space of the application. Figure 3 shows the pseudo code for the insert routine: lines 1 — 2 get an entry from the M-cache and set up the virtual address of the data to be protected, lines 3 — 5 set up the permissions metadata in the corresponding entry, and the final two instructions 6 — 7 try to update the metadata in the M-cache entry. An exception event between 1 — 6 (e.g., context switch) will cause line 7 to fail which restarts the routine.

2.4 Changing Permissions

The permissions metadata structure is in virtual memory space and this allows software to directly access and modify it. When changes to the access permissions (metadata) are made, we need to ensure that subsequent accesses to the data locations check the new permissions metadata. For example, assume there are two threads T1 and T2 on processors P1 and P2 that have write permissions on location A. If T1 decides to downgrade permissions to read-only, we need to ensure that (1) the old metadata entry in P2's M-cache is invalidated and (2) since P2 could have cached a copy of A in its L1 before the permissions changed, this copy is also invalidated. Both actions are necessary to ensure that P2 obtains new permissions on the next access. We deal with these in order.

Shooting down M-cache entries

This operation is simplified by the MPA field in the M-cache, which is used by hardware to ensure coherence of the cached metadata. These set of tags snoop on coherence events and any metadata updates result in invalidation of the corresponding M-cache entry. Hence all software has to do is update the metadata and the resulting coherence operations triggered will clean up all M-cache entries. The most straightforward option uses the physical address of the metadata as the MPA tag for the M-cache entry, but in essence software can set up any address. It is even valid and useful to have multiple M-cache entries managed by the same physical

address. For example, locations X and Y could each have an entry in the M-cache where the VA is tagged with the page addresses of X and Y while both their corresponding MPA entries could be tagged with MPA P. When P is written, a lookup in the MPA array would match two entries each with a back pointer to the corresponding VA entries of X and Y, which can then be invalidated — an efficient bulk invalidation scheme. Most previously proposed permission lookaside structures (e.g., TLB, Mondrian [24]) typically use interprocessor interrupts and sophisticated algorithms to shootdown the entry [19].

Checks of future accesses

Sentry moves permission checks to the L1 miss path and allows L1 hits to proceed without any intervention. Hence, when permissions are downgraded, the appropriate data blocks need to be evicted from all the L1 caches to ensure future accesses trigger permission checks. To cleanup the blocks from all L1s, software can perform prefetch-exclusives, which will result in invalidation of the cache blocks at remote L1s, and follow this by evicting the cache line from the local L1 (e.g., using PowerPC's dcb-flush instruction). A final issue to consider is that the cleanup races with other concurrent accesses from processors and the system needs to ensure that these accesses get the new permissions. To solve this problem, software must order the permission changes (including cleaning up remote M-cache entries) before the L1 cache cleanup so that subsequent accesses will reload the new permissions.

3. SENTRY PROTECTION MODELS

The protection framework in most previously proposed systems is dependent on hiding the protection metadata from manipulation by user-level applications. For example, the TLB, Mondrian, and Loki [24, 26] are all exclusively managed by low-level operating system software. To utilize the protection framework, every application in the system must cross protection boundaries via well-defined (system call) interfaces and abstractions to convey the application's protection policy to system software.

Our objective is to support low-cost and flexible protection models, with an emphasis on application-controlled intra-process protection. Hence, Sentry supplements the existing process-based protection provided by the TLB. It leaves the process-based protection untouched and permits relocation of the software routines that manage the M-cache to the user level (within each application itself). This eliminates the challenge of porting a new protection framework to the OS and reduces the risk of one application's policy affecting another application. Each application can independently choose whether to use the Sentry framework or not.

3.1 Foundations for Sentry Protection Models

A key concept in realizing protection is the *protection domain* — the context of a thread that determines the access permissions to each data location [10]. Every executing thread at any given instant belongs to exactly one protection domain and multiple threads can belong to the same protection domain. Furthermore, a thread can dynamically transition between different protection domains if the system policy permits it. Sentry uses integer values to identify protection domains. Domain 0 is reserved for the operating system while domain 1 and larger identifiers are used by applications. Within a process, different application domains must carry different identifiers, but domains in different processes may share the same identifier. Sentry is focused on intra-application protection and the M-cache entries are flushed on address space (process) context switches.

There are three fields that allow the hardware to recognize and restrict accesses based on domains: a per-thread `%Thread_Domain` register, a per-entry `Metadata_Domain` field in the M-cache, and a per-entry `Owner_Domain` field in the TLB (all illustrated earlier in Figure 2). `%Thread_Domain` is a new CPU register that identifies the protection domain of the currently executing thread. `Metadata_Domain` is a per-entry field in the M-cache identifying the protection domain that the entry's access permission information applies to. On an M-cache check, the `%Thread_Domain` register and the access address are both used to index into the M-cache. The M-cache entry with matching `Metadata_Domain` and virtual address tag is identified and its access permission information is then checked. The M-cache entries for a particular protection domain can be thought of as a capability list — they specify the data access permissions for a thread running in the domain. The ability to dynamically change a thread's protection domain gives software the flexibility to perform permissions changes over large regions without changing per-entry permissions.

The privilege of filling M-cache entries must be carefully regulated. If the M-cache were allowed to be modified by any domain, then a thread would be able to grant itself arbitrary permissions to any location. We introduce a per-entry `Owner_Domain` field in the TLB, which identifies the domain that "owns" the page corresponding to that entry. Only a thread in the page's owner domain or the exception handler in domain 1 can fill the M-cache for the locations in that page. The hardware enforces this by guaranteeing that an M-cache entry can be filled only when the `%Thread_Domain` register matches the target page's `Owner_Domain` [3] or when the `%Thread_Domain` is 1. Note that ownership is maintained at a coarser page granularity while the access control mechanism is managed at cache line granularity.

Domain 1 serves as an application-level supervisor, with the main thread of execution at the time of process creation being assigned to this domain. Specifically, Domain 1 enjoys the following privileges:

1. *Page Ownership :* Domain 1 controls page (both code and data) ownership; all requests for ownership change must be directed to the operating system via Domain 1.
2. *Thread Domain :* Domain 1 handles the tasks of adding, removing, and changing the domain of a thread during its lifetime.
3. *M-cache updates by non-owner domains :* M-cache exceptions (e.g., no metadata entry) triggered on accesses to non-owner locations (addresses owned by a different domain) are handled by Domain 1.
4. *Cross-Domain Calls :* Domain 1 ensures cross-domain calls (code that is in pages owned by a different domain) can occur only at specific entry points (according to application-specified policies registered with Domain 1).

In the following subsections we describe a few example protection models that can be realized using Sentry. We start from the simple case of one protection domain per process. Even in this degenerate case, we show Sentry's versatility in supporting low-cost watchpoints and protection of memory shared across OS processes. We then present a *compartment* model that can isolate various software modules and improve the safety of an application.

[3]The page table entry for the filled address needs to be in the TLB when filling the M-cache. If needed, we ensure that an M-cache fill instruction (see Table 1) triggers a TLB reload for the filled address.

3.2 One Domain Per Process

In this model, the existing process-based protection domain boundaries are inherited without any further refinement. Sentry's primary benefit is to support flexible cache-line granularity access control. All threads within a process belong to protection domain 1. The domain identifiers 0 and 1 differentiate between operating system and application. Sentry supports two modes of access control. The mode is determined by each page's owner domain (0 or 1) loaded into the TLB's `Owner_Domain` field. In the first mode (`Owner_Domain` is 1), the application threads retain ownership of all the pages, the privileges of filling the M-cache content, and handling permission exceptions. In the second mode (`Owner_Domain` is 0), only the operating system can perform these tasks.

The application-managed model incurs much less overhead than the OS-managed model. For instance, the cost of permission exception handling appears as an unconditional jump in the instruction stream (tens of cycles). In comparison, an OS-level exception incurs the cost of a privilege switch (hundreds of cycles), which requires switching privilege levels and saving registers. The application-managed model is useful for supporting cache-line-grain watchpoints. Reads and writes to specific locations can be trapped by setting appropriate M-cache permissions. The low-cost, cache-line-grain watchpoints can help with detecting various buffer overflow and stack smashing attacks, as well as with debugging code [15, 29]. This model has weak protection semantics since any part of the application can modify the M-cache.

In the OS-managed model, low-level system software directly manages the M-cache contents. This allows for a controlled sharing of cache-line granularity regions between processes. For example, in remote procedure calls (RPCs), the argument stack can be mapped between an RPC client and server process with different permissions (currently, a page granularity is used [1], incurring significant space or copy overhead). Sentry can be used to implement a safe kernel-user shared memory region without the need to ensure page alignment and can eliminate expensive memory copies between the operating system and applications.

3.3 Intra-Process Compartments

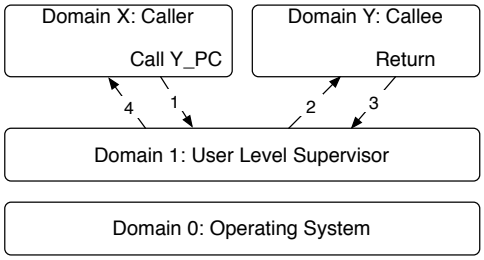
This model supports multiple protection domains within a single process. A real-world use of this model is the Apache application (see Figure 1) that supports various features (like caching pages fetched from the disk) in modules loaded into the web server's address space. The core set of developers set an interface between independently developed modules; isolating these modules into separate domains and enforcing the interface between them (data and functions) can improve reliability and safety.

In the compartment model, each domain owns a set of data and executable code regions. Threads in a protection domain can set up the permissions, execute code, and access the data owned by the domain. The domain can set up a handler to process M-cache misses and fill the M-cache.

Cross-Domain Data Accesses and Code Executions

Threads in a protection domain may need to access data that is not owned by the domain and we call these *cross-domain data accesses*. For instance, a web server module accesses some request data structure from the core application in order to apply output filters. Threads in a domain may also want to call functions that are not owned by the domain and we call these *cross-domain calls*. Individual domains may set up specific permissions for authorizing non-owner accesses and they must be carefully checked for safety.

One of our key design objectives is to minimize the role of the operating system and allow much of the management tasks to be performed directly by the application. This affords the most flexibility in terms of application-specific protection policies. It also incurs low management cost, since it keeps application-specific policies out of critical operating system components. Domain 1 in each process is designated as the user-level supervisor. Domain 1 is responsible for coordinating cross-domain data accesses and code invocation. It is implicitly given the ownership role for all memory in the process address space that the OS does not care about, and has the rights to fill M-cache content and handle M-cache permission exceptions. Domain 1 can also dynamically change the domain (anything other than domain 0) of a running thread by modifying the `%Thread_Domain` register. Given its privileges, domain 1 guards its own data and code against unauthorized accesses by owning those pages.



Steps of a cross-domain call when a thread in domain X invokes a function whose code is owned by domain Y:

1. Domain X tries to jump into a code region that it does not own. The hardware intercepts the call by recognizing a mismatch between the `%Thread_Domain` register and the `Owner_Domain` field in the TLB entry associated with the target instruction. The target instruction `Y_PC` is saved. The hardware then effects an exception into the `%Domain1_Handler` while simultaneously changing the current domain to domain 1.

2. If the target address has been registered as a cross-domain call, the domain 1 exception handler marshals the arguments on the stack, grants the appropriate data access permissions to the callee domain, and invokes the function through a special `switch_call` instruction. The instruction jumps into `Y_PC` and changes the protection domain context (`%Thread_Domain`) to Y.

3. On return, the function tries to jump back to the caller. Since the return target is owned by domain X, an exception is triggered into domain 1 and the return target address is saved.

4. Domain 1 finally executes `switch_call` back to X.

Figure 4: Cross-domain call execution flow.

Non-owner data accesses that trigger M-cache miss exceptions (i.e., no M-cache entry) need to be handled carefully. The thread itself lacks the privilege to set up the metadata. Hence, these operations are handled by domain 1. The address to this exception handler in domain 1 is located in a CPU register (`%Domain1_Handler`), which is managed exclusively by domain 1. On a non-owner M-cache exception, hardware changes the domain context (i.e., %Thread_Domain register) to domain 1 and traps to the `%Domain1_Handler`. Based on application specific policies, the domain 1 exception handler can let the non-owner data access progress (finish or raise an exception) by filling the ap-

propriate M-cache entry. When the miss handler has finished its operations, domain 1 reverts back to the original domain context to continue execution. Subsequent accesses to the same location (from the non-owner domain) can proceed without interruption.

Cross-domain calls are registered with domain 1. A cross-domain call and its return are redirected and managed by domain 1 in a four-step process illustrated in Figure 4. The indirection through domain 1 for cross-domain calls does impose a minor performance penalty. Unlike earlier work that speeds up cross-address space calls by using shared-memory segments [1], our cross-domain call remains in the same address space and the complete execution flow utilizes the same stack. This allows us to achieve efficiencies comparable to a function call.

4. EVALUATIONS

Our base hardware is a 16-core chip (1 GHz clock frequency), which includes private L1 instruction and data write-back caches (32KB, 4-way, 1 cycle) and a banked, shared L2 (8MB, 8-way, 30 cycle), with a memory latency of 300 cycles. The cores are connected to each other using a 4×4 mesh network (3 cycle link latency, 64 byte links). The L2 is organized on one side of this mesh and the cores interface with the 16-banked L2 over a 4×16 crossbar switch. The L1s are kept coherent with the L2 using a MESI directory protocol. This coherence protocol is based on the SGI ORIGIN 3000 3-hop scheme with silent evictions. Our infrastructure is based on the full-system GEMS-Simics framework. We faithfully emulate the Sentry hardware/software interface and model the latency penalty of the software handlers from within the simulator. We simulate all memory references made by the handler and use instruction counts to estimate the latency of non-memory instructions.

We begin by evaluating the M-cache implementation (area and latency) cost and basic management costs (software and hardware). We then evaluate the use of Sentry for three applications: Apache, remote procedure call (RPC), and memory debugging. Via the Apache and RPC benchmarks, we demonstrate the flexibility, usability, and performance efficiency of utilizing Sentry to implement application-level protection. In evaluating Sentry for debugging, we compare its performance overheads against MemTracker [21] and Mondrian [24].

4.1 M-cache Implementation

Area, Latency, and Energy

The M-cache area implications are a function of its size and organization. All known cache and TLB optimizations apply (banking, associativity, etc.). Most importantly, the M-cache intercepts only L1 misses, thereby reducing its impact on the processor critical path. While dual-tagged, each tag array is single-ported. The virtual tag is accessed only from the processor side (for checking permissions and filling M-cache entries) and the MPA is accessed only from the network side (for snoop-based coherence). We used CACTI 6 [13] to estimate cycle time for a 256-entry M-cache (4KB, 4 way, 16 bytes per entry) that provides good coverage (fine-grain permissions for a 1MB working set). We estimate that in 65 nm high-performance technology (ITRS HP), the M-cache can be indexed within 0.5ns (1 processor cycle using our parameters). Furthermore, since the M-cache is located outside the processor core and is accessed in parallel with a long latency L1 miss we can trade latency for energy benefits. If we use the low power transistor technology in CACTI (ITRS LOP) , the access latency increases by 2-3$\times$ (to $\sim$3 cycles) compared to high-performance transistor technology (ITRS HP). There is a significant energy reduction: an M-cache with ITRS LOP consumes 1% of the leakage power and

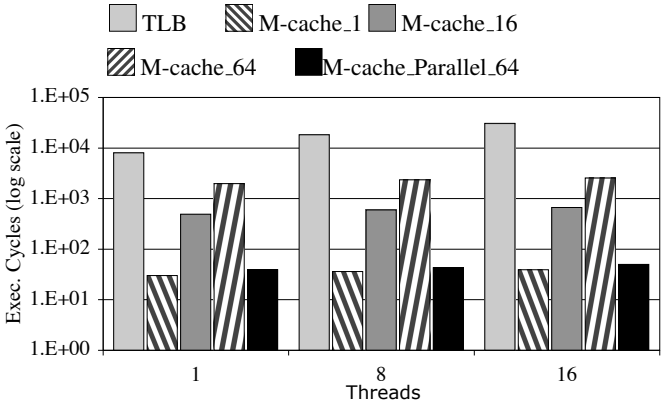


Figure 5: TLB vs. M-cache. M-cache_N measures execution cycles for changing permissions on N cache lines. L1 handles stores in order. In M-cache_Parallel_64, the L1 can sustain 64 concurrent misses.

33% of the dynamic access power of an M-cache employing ITRS HP. On average (geometric mean), across the PARSEC, SPLASH2, and Apache workloads, the M-cache with ITRS LOP consumes 0.029nJ, 0.037nJ, and 0.27nJ respectively per operation; in comparison a TLB with the same number of entries consumes 3nJ.

Operation cost

There are two main management costs associated with any hardware protection mechanism: (1) the cost of switching to the privilege level that can manipulate it and (2) the cost of maintaining the coherence of the cached copies of the metadata. The M-cache can be managed entirely at user level. In addition, the metadata physical address (MPA) tag allows the M-cache to exploit coherence to propagate metadata invalidations to remote processors. This simplifies the software protocol required to manage the M-cache and improves performance compared to existing protection mechanisms that use the interprocessor interrupt mechanism.

We compare the cost of changing the metadata associated with the M-cache against the cost of manipulating page-attribute protection bits in the TLB. We set up a microbenchmark that creates K threads (varied from 1—16) on the machine, and every 10,000 instructions a random thread is picked to make permission changes. To test the TLB, we change permissions for a page, and to test the M-cache we change permissions for N cache lines within a page. Overall, modifying permissions with the M-cache is 10—100$\times$ faster than TLB shootdowns (see Figure 5). The dominant cost with the M-cache is that of purging the data from the L1 caches (see Section 2.4). The routine needs to prefetch the data cache blocks in exclusive mode in order to invalidate all L1s, and this typically results in coherence misses. In our design, each L1 cache allows only one outstanding miss and hence the invalidation of each cache line directly appears on the critical path. The latency of the permission change is directly proportional to the number of cache blocks that are invalidated (M-cache_1, M-cache_16 and M-cache_64 bars). We also evaluate performance when 64 outstanding L1 prefetches are allowed (M-cache_Parallel_64) and show that overlapping the latency of multiple misses is sufficient to significantly reduce permission change cost.

Teller et al. [19] discuss hardware support to keep TLBs coherent and recently, concurrent with our work, UNITD [16] explored the performance benefits of coherent TLBs. Both these works mainly seek to reduce the overheads of conventional OS TLB management routines while Sentry employs coherence to enable user-level software routines to manage the M-cache.

4.2 Compartmentalizing Apache

In this section, we use Sentry's intra-process compartment protection model (see Section 3.3) to enforce a safety contract between a web server (Apache) and its modules. Apache includes support for a standard interface to aid module development; the "Apache Portable Runtime" (APR) [31] exports many services including support for memory management, access to Apache's core data structures (e.g., file stream), and access to intercept fields in the web request. Apache's modules are typically run in the same address space as the core web server in the interest of efficiency and the desire to maintain a familiar programming model (conventional procedure calls between Apache and the module). A module therefore has uninhibited access to the web server's data structures, resulting in the system's safety relying on the module developers' discipline.

Our goal is to (1) isolate each module's code and data in a separate domain and ensure that the APR is enforced. This protects the web server's data and ensure that modules can access the web server's data only through the exported routines; and (2) achieve this isolation with simple annotations in the source code without requiring any source revisions. While the definition of a module may be abstract, here we use the term to refer to the collection of code that the developers included to extend the functionality of the web server. To enable protection, Sentry annotates the source to perform the following tasks: (1) specify the ownership (domain) of code and data regions, (2) assign permissions to the various data regions for the different domains, and (3) assign domains to preforked threads that wait for requests. To simplify our evaluation, we set up the core Apache code (all folders other than module/) to directly execute in domain 1 and emulate all the actions required by domain 1 (those that would be provided in the form of a library) for cross-domain calls from within the simulator. The only modifications required to Apache's source code were the instructions that set up the domains and permissions.

The modules we compartmentalized are `mod_cache` and `mod_mem_cache`, which work in concert to provide a caching engine that stores pages in memory that were previously fetched from disk. "mod_cache" consists of two parts: (1) module-Apache interface (`mod_cache.c`) and (2) the cache implementation (`cache_cache.c[.h]`,`cache_hash.c[.h]`, and `cache_pqueue.c[.h]`). "mod_cache" also needs to interface with a storage engine, for which we use `mod_mem_cache` (`cache_storage.c, mod_mem_cache.c`). We compiled the modules into the core web server and compartmentalized the storage engine (mod_mem_cache) into domain 2 and the cache engine (mod_cache) into domain 3, respectively.

Compartmentalizing Code

Our primary goal was to enforce the APR interface and ensure that module domains cannot call into non-APR functions. First, we compiled in the modules and did a static object code analysis to determine the module boundaries. We then added routines to the core Apache web server that (1) registered the APR code regions as owned by Apache and set up read-write permissions for Apache (domain 1) and read-only permissions for the modules and (2) declared the individual module code regions as owned by the appropriate domains. The modules expose only a subset of the functions to other modules (e.g., the storage module exports the `cache_create_entity()` to the cache module as read-only while it hides the internal `cleanup_cache_mem`, which is used for internal memory management). The module's entire code is accessible to the Apache web server. Finally, the pre-forked worker threads are started in domain 1. When the threads call into a module's routines, the exception handler in domain 1 transitions the thread to the appropriate domain if the call is made to the module's exported APR. Non-APR calls from the module would be caught by Sentry.

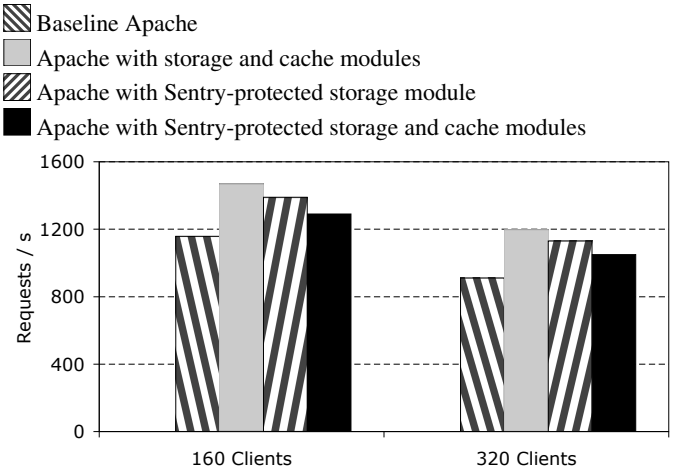


Figure 6: Performance of Sentry-protected Apache with compartmentalized modules.

Since memory ownership is assigned at the page granularity, we have to ensure that code from two different modules or the core web server are not linked into the same page. While current compilers do not provide explicit support for page alignment, it is fortunate that our compiler (gcc) allows code generated from an individual source file to remain contiguous in the linked binary. Given the contiguity of each module's code segment, we added appropriate padding in the form of no-ops (`asm volatile nops`) to the end of the source (.h and .c files) to page align each module.

Compartmentalizing Data

Assigning ownership to data regions proved to be a simpler task. The core web server and modules all use the APR provided memory allocator (`srclib/apr/memory/unix/apr_pools.c`). With this allocator, it was possible to set up different pools of memory regions and request allocation from a specific pool. We specified separate pools for each of the domains: Apache web server (domain 1), mod_mem_cache module (domain 2), and mod_cache module (domain 3). We then assigned ownership of the pool to the domain that it served. The allocator itself is part of the APR interface (domain 1). The permissions rules we set up were (1) Apache's core (domain 1) can read/write all memory in the program, (2) each module can read/write any memory in their pool, and (3) a module has read-only access to some specific variables exported by other external modules — this is where fine-grain permission assignment was useful. In some cases, a variable exported by a module (read-only for remote domains) was located in the same page as a local variable (no permissions). For example, the cache object in the storage engine (mem_cache_object) contained two different sets of fields, those that described the page being cached (e.g., html headers) and those that described the actual storage (e.g., position in the cache). The former needs to be readable from the cache engine, while the latter should be inaccessible.

Performance Results

We now estimate the overheads of compartmentalizing the modules in Apache. In this experiment, we pre-compiled modules into the core kernel, disabled logging, and used POSIX mutexes for synchronization. The specific configuration we used is *-with-mpm=worker -enable-proxy -disable-logio -enable-cgi -enable-*

cgid -enable-suexec –enable-cache. We configured the Surge client for two different loads: 10 or 20 threads per processor (total of 160 or 320 threads) and set the think time to 0ms. Our total webpage database size was 1GB with the file sizes following a Zipf distribution (maximum file size of 10 MB). We warmed up the software data structures (e.g., OS buffer cache) for 50,000 requests and measured performance for 10,000 requests. Sentry permits incremental adoption of compartment-based protection. In our experiments we protect apache's modules in two stages: we first moved `mod_mem_cache` into a separate domain leaving `mod_cache` within domain 1 (along with the core webserver). Subsequently, we moved both `mod_mem_cache` and `mod_cache` out of domain 1, each into a separate domain.

Figure 6 shows relative performance. Sentry protection imposes an overhead of ≃13% when compartmentalizing both `mod_cache` and `mod_mem_cache` and a ≃6% when compartmentalizing just the `mod_mem_cache`. The primary overheads with Sentry are the execution of the handlers required to set up data permissions in the M-cache and the indirection through domain 1 needed for cross-domain calls. Approximately 20% of the functions invoked by the module are APR routines, which involve a domain crossing. We believe the overhead is acceptable given the safety guarantees and the level of programmer effort needed.

Process-based Protection vs. Sentry

To evaluate the protection schemes afforded by OS process-based protection, we develop mod_case (a module that changes the case of ASCII characters in the webpage) using process-based protection and compare it against Sentry's compartment model. To employ process-based protection we needed to make significant changes to the programming model and reorganize the code and data. We had to implement a mod_case specific shim (implemented as an Apache module) that communicates with mod_case through IPC (inter-process communication), requiring a shared-memory segment. All interaction between Apache and mod_case passes through the shim, which converts the function calls between Apache and mod_case into IPC. Although Apache's APR helps with setting up shared-memory segments between the shim and the mod_case process, the shim and the module still needed to write explicit memory routines (for each interface function) to copy from and to the shared argument passing region. The conversion of even this simple module to use IPC was frustrated by the inability to pass data pointers directly between the processes and the need to use a specific interface between the shim and the module itself. As Figure 7 shows, the process-based protection adds significant overhead (33%) compared to Sentry (7%). To summarize, we believe the programming rigidity, need to customize the interface for individual modules, and performance overheads are major barriers to adoption of process-based protection.

We briefly compare Sentry's cross domain calls against existing IPC mechanisms. Despite considerable work to reduce the cost of IPC [1, 8, 12], it is still 10^3 slower than an intra-process function call. In contrast, Sentry focuses on functions calls between protection domains established within a process. They have lower overheads since they share the OS process context and crossing domains does not involve address-space switches. The indirection through the user-level supervisor domain 1 adds 2 call instructions (to jump from caller→ to domain 1 and then into callee) and 9 instructions to align the parameters to a cache line boundary and set appropriate permissions to the stack for the callee domain (i.e., permissions to access only the parameters and its own activation records). The overhead at runtime varies between 20 and 30 cycles (compared to 5 cycles required for a typical function call).

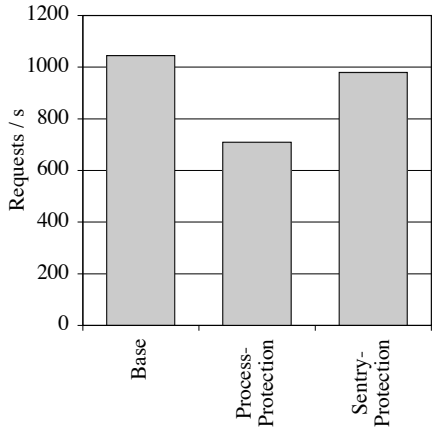


Figure 7: Comparing process-based protection with Sentry. Base: Apache with mod_case.

4.3 Lightweight Remote Procedure Call

RPC is a form of inter-process communication where the caller process invokes a handler exported by a callee process. We first describe the scheme used to implement RPC in current OS kernels (e.g., Solaris): (1) The caller copies arguments into a memory buffer and generates a trap into the OS. (2) The OS copies the buffer content into the address space of the callee. It then context switches into the callee. (3) To return, the callee traps into the kernel, which unblocks the caller process. The main overhead is due to the four privilege switches (two from user→kernel space and two from kernel→user space) and the memory copy required. Earlier proposals [1] have optimized RPC by using a shared memory segment (i.e., `shmem()`) to directly pass data between the caller and callee process. The minimum size of the shared memory segment is a page and a copy is still needed from the argument location to the shared page. More recently, Mondrian [24] postulated that word granularity permission and translation can eliminate the argument passing region.

We use the M-cache to provide fine-grain access control of the locations that need to be passed from the the caller to the callee process and eliminate the copying entirely. The caller process must align the arguments to a cache line boundary and request the kernel to map the argument memory region into the callee's address space (requiring a user-kernel and kernel-user crossing). We experiment with a client that makes RPC calls to a server periodically (every 30,000 cycles). The server is passed a 2 KB random alphabet string that it reverses while switching the case of characters. We compare the cost of argument passing using several approaches — Sentry, the implementation in the Solaris RPC library, and an optimized implementation that uses page sharing [1]. Our results indicate that completely eliminating the copying provides a (∼9–10×) speedup compared to the optimized page sharing approach. The unoptimized RPC implementation has 10^3–10^4 × higher latency.

4.4 Debugger: Sentry-Watcher

We developed Sentry-Watcher, a C library that applications call into for watchpoint support. A watchpoint is usually set by the programmer to monitor a specific region of the memory and when an access occurs to this region, it raises an exception. Sentry-Watcher employs the one-domain per process model and operates in the application mode (permissions metadata and M-cache managed by a library in the application space). It exports a simple interface: `insert_watchpoint()`, `delete_watchpoint()`, and `set_handler()`, with which the programmer can specify

the range to be monitored, type of accesses to check, and the handler to trigger.

Apart from fine granularity watchpoints, there are three additional benefits with Sentry-Watcher: (1) it supports flexible thread-specific monitoring, allowing different threads to set up different permissions to the same location [4], (2) it supports multi-threaded code efficiently since the hardware watchlist can be propagated in a consistent manner across multiple processors at low overhead (also supported by MemTracker [21]), and (3) it effectively virtualizes the watchpoint metadata and allows an unbounded number of watchpoints.

Benchmark	Mallocs/1s	Heap Size	Heap Access %	Bug Type
BC	810K	60KB	75%	BO
Ncom	2	8B	0.6%	SS
Gzip	0	0	0%	BO,IV
GO	10	294B	1.9%	BO
Man	350K	210KB	89%	BO,SS
Poly	890	11KB	27%	BO
Squid	178K	900KB	99.5%	ML

B-Bytes, KB-Kilobytes, MB-Megabytes
BO-Buffer Overflow SS-Stack Smash, ML-Memory Leak, IV-Invariant Violation. Squid is multi-threaded.

Table 2: Application Characteristics

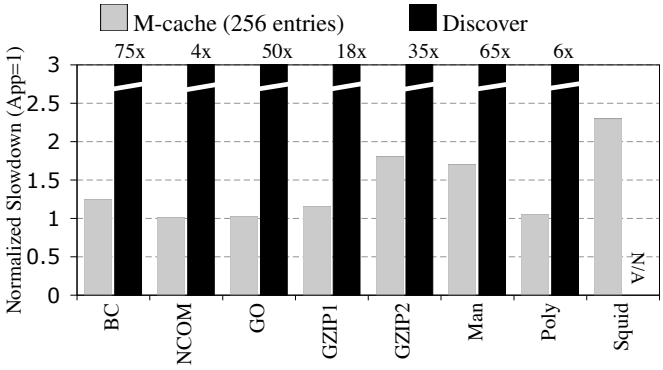


Figure 8: Sentry-Watcher vs. Binary instrumentation. N/A - Discover is not compatible with multi-threaded code. Gzip1 detects Buffer-Overflow bugs. Gzip2 detects memory leak, buffer overflow, and stack smash bugs.

Debugging Examples

Generic unbounded watching of memory can help widen the scope of debugging. Here, we use the system to detect four types of bugs: (1) Buffer Overflow — this occurs when a process accesses data beyond the allocated region. To detect it we pad all heap buffers with an additional cache line and watch the padded region. (2) Stack Smashing — A specific case of buffer overflow where the overflow occurs on a stack variable and manages to modify the return address; we watch the return addresses on the stack. Dummy arguments need to be passed to the functions to set up the appropriate padding and eliminate false positives. (3) Memory Leak — We monitor all heap allocated objects and update a per-location timestamp on every access. Any location with a sufficiently old access timestamp is classified as a potential memory leak. (4) Invariant Violation—Monitor application-specific variables and check specified assertions. We demonstrate the prototype on the benchmark suite provided with iWatcher [29] — Table 2 describes the benchmark properties. We compare the performance of Sentry-Watcher against Discover (a SPARC binary instrumentation tool [32]). Sentry-Watcher is evaluated on our simulator. Discover is set up on the Sun T1000. Compared to the binary instrumentation technique, Sentry-Watcher provides 6–75× speedup. Sentry still incurs noticeable overhead compared to the original applications—varying between 3–50% for most applications. At worst, we encounter up to ∼2× overhead on the memory-leak detector experiments, which instrument all heap accesses (see Squid in Figure 8).

[4]Since all threads run in the same domain, this would require a flush of the M-cache on a thread switch.

Comparing with Other Hardware

To understand the performance overheads of Sentry-Watcher, we further compared it against the performance of MemTracker [21], a hardware mechanism tuned for debugging, and Mondrian [24], a fine- and variable-grain flexible protection mechanism placed within the processor pipeline.

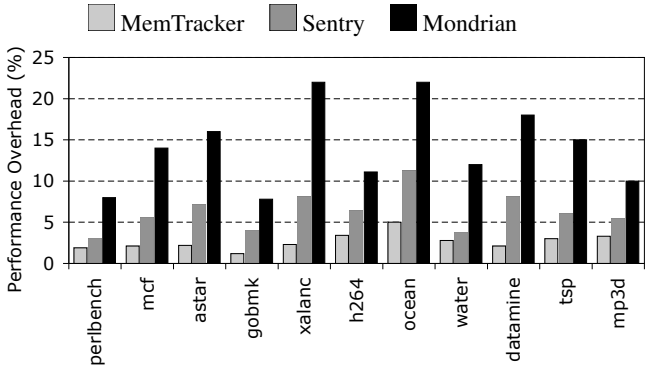


Figure 9: Effect of software handlers and cost of manipulating metadata in the OS.

In MemTracker, a software-programmable hardware controller maintains the metadata, fetching and operating on it in parallel with the data access. To emulate the controller's operations, we assign a 0 cycle penalty for all operations other than the initial setup of the metadata. This is an optimistic assumption since typically metadata misses in MemTracker also add to the overhead. In Mondrian, setting up or changing the metadata require privilege mode switches into and out of the operating system kernel. To estimate the cost of a fast privilege switch, we measure the latency of an optimized low overhead call (e.g., `gethrtimer()`) on three widely available microarchitectures (SPARC, x86, and PowerPC). We observed 280 cycles (SPARC), 312 cycles (x86) and 450 cycles(PowerPC). To emulate Mondrian, we add a penalty of 300 cycles to every permission exception and metadata modification; we assign a 0 cycle penalty to metadata operations. To limit the number of variables, we keep the metadata cache size fixed across all the systems (256 entries). For this comparison, we implemented the debugger tool discussed by the MemTracker paper [21], which checks for heap errors, buffer overflow, and stack smashing. The workloads we use are from the SPECCPU 2006 suite [21]. We also include the SPLASH2 benchmarks to verify that our findings are valid for multi-threaded workloads. Figure 9 shows that the overhead of Sentry-Watcher averages 6.2%, compared to the idealized MemTracker's 2.6% and Mondrian's 14%. Since MemTracker requires a hardware controller to fetch and manipulate the metadata while we leave all such operations in software, we believe that our system is more complexity-effective.

5. RELATED WORK

5.1 Access Control for Data Protection

Access control mechanisms are commonly exploited to protect data against inappropriate accesses from buggy programs or malicious attacks. Most modern processors and operating systems associate a protection domain (or a process) with a separate, linear page-based address space. Multiple threads in a process typically see the same access permission to a location. With the coupling of the protection domain and address space, an inter-domain communication requires operating system intervention and pointer swizzling [23]. Furthermore, domains are assigned at process creation and persist for a process' lifetime. OS processes are widely used in fault isolation applications to sandbox software components [25]. Unfortunately, processes also impose complete isolation between sandboxes and the programmer has to use custom library interfaces to enable data sharing.

Efforts were made to separate protection from the address space using a protection lookaside buffer [9] and page tagging labels and protection identifiers (in HP's PA-RISC [22]). These designs are confined to a page or larger granularity and cannot provide the fine granularity of control over protection needed by current modular software. While x86 segmentation hardware [4] allows variable granularity, the number of segment registers available is small. Moreover, the PA-RISC and x86 hardware hardcode the relationship among domains. Sentry is highly flexible and allows each application to choose its protection model independently.

Several approaches specifically target protection within the operating system but they lack the flexibility to support application-level protection. For instance, SPIN takes advantage of type-safe languages in constructing safe OS extensions [2]. The required use of certain type-safe languages can restrict general application development. As another example, Nooks uses lightweight processes and page protection to guard the kernel against device driver bugs [18]. Unfortunately, when data is shared between domains, these schemes require the programmer to use custom wrapper libraries to enable the protection framework to capture and marshal the accesses.

Capability systems [3,5,11,17] augment object reference pointers to include information on access control. The capabilities shared between threads are marshaled by the OS and can support generalized protection models. Typical capability implementations change the layout of memory and fatten pointers. Software developers need to be aware of the modified layout and typically need code rewrites, which lessens their appeal. More importantly, the relatively large management cost for a capability (e.g., when revoking access rights) makes it ill-suited to protecting fine-grain data elements. Typically, capability-protected objects are external resources such as files, or memory segments at large granularity.

Recently, Mondrian [24] decoupled protection from a conventional paging framework and implemented it using segments. An application's address space is described by a collection of variable-sized segments, each capable of supporting word granularity. This flexibility comes at the cost of additional hardware and operating system modifications. This work replaces the existing protection framework (TLB) with a new permissions-lookaside-buffer (PLB) that checks all accesses in the pipeline and needs add-ons (e.g., sidecar registers) to reduce the performance overheads. Furthermore, it introduces new layers in the operating system to implement all protection (intra-process and inter-process) based on the PLB approach. This requires every application to communicate its policy to the low-level supervisor.

Loki [26] adopted a different tagging strategy, choosing to tag physical memory with labels that further map to a set of permission rules. Loki allows system software to translate application security policies into memory tags. Threads must have the corresponding entry in the permissions cache in order to be granted access permission. If the threads need to be segregated into separate domains, then this would require software support to convert inter-thread function calls into cross-domain call gates. Permission cache modifications must be performed within the operating system. Permission revocation in the case of page swapping would require a sweep of all process/thread's permissions caches.

Our Approach

While we share the goals of Mondrian and Loki to allow more flexible protection models, Sentry employs an auxiliary protection architecture that supplements existing mechanisms on commodity platforms. Specifically, Sentry's hardware is implemented entirely outside the processor core, is subordinate to the existing TLB mechanism, and intercepts only L1 misses. Sentry uses coherence states to implicitly verify all L1 hits, which saves the energy and performance overhead of checking the metadata for a large fraction of the accesses. Further, it also maintains coherence of the protection metadata across multiple processors, simplifying software management (no need for interprocessor interrupts to ensure consistency of the metadata). Finally, no changes are made to the existing process-based protection and the intra-process protection models may be implemented at the user level. Compared to Mondrian and Loki, Sentry reduces the changes required to the core hardware and operating system software. It enables flexible, low-cost protection within individual applications. Finally, it incurs space or time overhead only when auxiliary intra-application protection is needed.

5.2 Access Control for Debugging

When used for debugging applications, the space and performance overheads are directly proportional to the number of locations being watched. Prior proposals mainly focus on reducing the overhead of intercepting memory operations and manipulating some debugger-specific metadata. This capability is sufficient to detect a variety of memory bugs [14].

Hardware proposals seek to eliminate the performance overheads of software-based debugging by intercepting and checking accesses in a transparent manner. A common feature of all these works is that they intercept all loads and stores in the processor pipeline. They also share the metadata among different threads, which makes it difficult to set up thread-specific access control. The major differences among these systems is the hardware state bits used to track the watchpoint metadata: bloom filters [20], additional cache-tag bits [30], ECC bits [15, 29], or separate lookaside structures [21]; which have varying levels of false positives and coverage characteristics. They also vary in whether the hardware semantics are hardcoded for a specific tool [28] or support a general-purpose state machine [21].

Our Approach

The M-cache mechanism proposed in this paper is a more complexity-effective implementation since it resides entirely outside the processor core and intercepts only L1 cache misses. It uses cache coherence to enable low-cost maintenance of the metadata scattered across multiple processors (without requiring hardware controllers to coordinate data and metadata movement [21]). Furthermore, it supports thread-specific monitoring, allowing different threads to have different permissions to the same location.

6. CONCLUSIONS

Light-weight and flexible data protection is highly desirable for the improved reliability and debugging support of modular software. Sentry provides a protection framework using virtual memory tagging that is completely subordinate to the existing OS-TLB framework. The main hardware component, the metadata cache (M-cache), implements permission checks entirely outside the processor core. It intervenes only on L1 misses, transparently reusing the L1 coherence states to enforce permissions on hits. Compared to previously proposed protection hardware that intercepts all accesses in the pipeline, the M-cache is a complexity-effective design that provides significant energy reduction (consuming 1—10% of the energy of the in-core approach).

Sentry realizes intra-application protection models with predominantly user-level software, requires very little OS intervention, and makes no changes to process-based protection. We developed an intra-application compartment protection model and used it to isolate the modules of a popular web server application (Apache), thereby protecting the core web server from buggy modules. Our evaluation demonstrated that Apache's module interface can be enforced at low overhead ($\simeq$13%), with few application annotations, and in an incremental fashion. Finally, the user-level management of Sentry also supports low-overhead, fine-grain watchpoints. We developed a memory debugger to demonstrate that Sentry's flexibility and overheads are comparable to previously proposed hardware-based debuggers. We show that user-level management (as opposed to OS-level management) is an important design choice to limit the overheads in such applications.

7. REFERENCES

[1] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight Remote Procedure Call. *ACM Trans. on Computer Systems*, 8(1):37-55, Feb. 1990. Originally presented at the *12th ACM Symp. on Operating Systems Principles*, Dec. 1989.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. . Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, Dec. 1995.

[3] N. P. Carter, S. W. Keckler, and W. J. Dally. Hardware Support for Fast Capability-Based Addressing. In *Proc. of the 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.

[4] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, Dec. 1999.

[5] S. Colley, G. Cox, K. Lai, J. Rattner, and R. Swanson. The Object-Based Architecture of the Intel 432. In *Proc. of the IEEE COMPCON Spring '81*, Feb. 1981.

[6] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: software guards for system address spaces. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, 2006.

[7] J. R. Goodman. Coherency for Multiprocessor Virtual Address Caches. In *Proc. of the 2nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1987.

[8] F. Kaashoek, D. Engler, G. Ganger, H. Briceno, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, Oct. 1997.

[9] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architectural Support for Single Address Space Operating Systems. In *Proc. of the 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992.

[10] B. W. Lampson. Protection. In *Proc. of the 5th Princeton Symp. on Information Sciences and Systems*, pages 437-443, Mar. 1971. Reprinted in *ACM SIGOPS Operating Systems Review 8*:1 (Jan. 1974), pp. 18-24.

[11] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, MA, 1984.

[12] J. Liedtke. Improving IPC by Kernel Design. In *Proc. of the 14th ACM Symp. on Operating Systems Principles*, Dec. 1993.

[13] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches With CACTI 6.0. In *Proc. of the 40th Intl. Symp. on Microarchitecture*, Dec. 2007.

[14] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of the SIGPLAN 2007 Conf. on Programming Language Design and Implementation*, June 2007.

[15] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proc. of the 10th Intl. Symp. on High Performance Computer Architecture*, Feb. 2005.

[16] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy. UNified Instruction/Translation/Data (UNITD) Coherence: One Protocol to Rule Them All. In *Proc. of the 16th Intl. Symp. on High Performance Computer Architecture*, Jan. 2010.

[17] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a Fast Capability System. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, Charleston, SC, Dec. 1999.

[18] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, Oct. 2003.

[19] P. J. Teller, R. Kenner, and M. Snir. TLB Consistency on Highly-Parallel Shared-Memory Multiprocessors. In *Proc. of the 21st Hawaii Intl. Conf. on System Sciences*, Jan. 1988.

[20] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas. SoftSig: software-exposed hardware signatures for code analysis and optimization. In *Proc. of the 13th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar. 2008.

[21] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. MemTracker: Efficient and Programmable support for Memory Access Monitoring and Debugging. In *Proc. of the 13th Intl. Symp. on High Performance Computer Architecture*, Feb. 2007.

[22] J. Wilkes and B. Sears. A Comparison of Protection Lookaside Buffers and the PA-RISC Protection Architecture. HLP-92-55, Hewlett Packard Laboratories, Mar. 1992.

[23] P. R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *Proc. of the Intl. Workshop on Object Orientation in Operating Systems*, Sept. 1992.

[24] E. Witchel, J. Cates, and K. Asanović. Mondrian Memory Protection. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[25] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy*, May 2009.

[26] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware Enforcement of Application Security Policies. In *Proc. of the 8th Symp. on Operating Systems Design and Implementation*, Dec. 2008.

[27] Q. Zhao, R. M. Rabbah, S. P. Amarasinghe, L. Rudolph, and W.-F. Wong. How to Do a Million Watchpoints: Efficient Debugging Using Dynamic Instrumentation. In *Springer-Verlag*, 2008.

[28] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-based Invariants. In *Proc. of the 37th Intl. Symp. on Microarchitecture*, Dec. 2004.

[29] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architecture Support for Software Debugging. In *Proc. of the 31st Intl. Symp. on Computer Architecture*, June 2004.

[30] UFO: A General-Purpose User-Mode Memory Protection Technique for Application Use UIUCDCS-R-2007-2808, Jan 2007.

[31] Apache Project. In *http://Apr.apache.org/*.

[32] Cool Tools. In *http://cooltools.sunsource.net/*.