Mondrian Memory Protection

Emmett Witchel, Josh Cates, and Krste Asanović
MIT Laboratory for Computer Science, Cambridge, MA 02139
{witchel,cates,krste}@lcs.mit.edu

ABSTRACT

Mondrian memory protection (MMP) is a fine-grained protection scheme that allows multiple protection domains to flexibly share memory and export protected services. In contrast to earlier page-based systems, MMP allows arbitrary permissions control at the granularity of individual words. We use a compressed permissions table to reduce space overheads and employ two levels of permissions caching to reduce run-time overheads. The protection tables in our implementation add less than 9% overhead to the memory space used by the application. Accessing the protection tables adds less than 8% additional memory references to the accesses made by the application. Although it can be layered on top of demand-paged virtual memory, MMP is also well-suited to embedded systems with a single physical address space. We extend MMP to support segment translation which allows a memory segment to appear at another location in the address space. We use this translation to implement zero-copy networking underneath the standard `read` system call interface, where packet payload fragments are connected together by the translation system to avoid data copying. This saves 52% of the memory references used by a traditional copying network stack.

1. INTRODUCTION

Operating systems must provide protection among different user processes and between all user processes and trusted supervisor code. In addition, operating systems should support flexible sharing of data to allow applications to co-operate efficiently. The implementors of early architectures and operating systems [5, 26] believed the most natural solution to the protected sharing problem was to place each allocated region in a segment, which has the protection information. Although this provides fine-grain permission control and flexible memory sharing, it is difficult to implement efficiently and is cumbersome to use because each address has two components: the segment pointer and the offset within the segment.

Modern architectures and operating systems have moved towards a linear addressing scheme, in which each user process has a separate linear demand-paged virtual address space. Each address space has a single protection domain, shared by all threads that run within the process. A thread can only have a different protection domain if it runs in a different address space. Sharing is only possibly at page granularity, where a single physical memory page can be mapped into two or more virtual address spaces. Although this addressing scheme is now ubiquitous in modern OS designs and hardware implementations, it has significant disadvantages when used for protected sharing. Pointer-based data structures can be shared only if the shared memory region resides at the same virtual address for all participating processes, and all words on a page must have the same permissions. The interpretation of a pointer depends on addressing context, and any transfer of control between protected modules requires an expensive context switch. The coarse granularity of protection regions and the overhead of inter-process communication limit the ways in which protected sharing can be used by application developers. Although designers have been creative in working around these limitations to implement protected sharing for some applications [9], each application requires considerable custom engineering effort to attain high performance.

We believe the need for flexible, efficient, fine-grained memory protection and sharing has been neglected in modern computing systems. The need for fine-grained protection in the server and desktop domains is clear from the examples of a web server and a web client. These systems want to provide extensiblility where new code modules can be linked in to provide new functionality. The architects of these systems have rejected designs using the native OS support for a separate address space per module because of the complexity and run-time overhead of managing multiple address contexts. Instead, modern web servers and clients have solved the extensiblity problem with a plugin architecture. Plugins allow a user to link a new module into the original program to provide a new service. For instance, the Apache web server has a plugin for the interpretation of perl code in web pages [2], and browsers support plugins to interpret PDF documents [1]. Linking in code modules makes communication between the server and the plugin fast and flexible, but because there is no protection between modules in the same address space it is also unsafe. Plugins can crash an entire browser, or open a security hole in a server (e.g., from a buffer overrun).

Embedded systems have the same problem since they are often organized as a set of tasks (sometimes including the operating system) that share physically-addressed memory (see Section 7). Without inter-task protection, an error in part of the system can make the entire system unreliable. Similarly, loadable OS kernel modules (such as in Linux) all run in the kernel's unprotected address space, leading to potential reliability and security problems.

Figure 1 illustrates a general protection system and is based on the diagrams in [17] and [18]. Each column represents one *protection domain* [16] while each row represents a range of memory ad-

ASPLOS-X '02 San Jose, CA

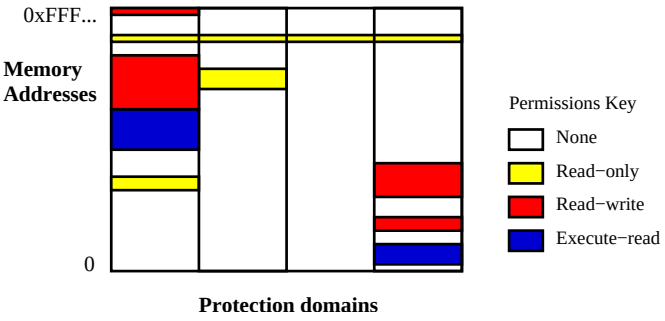


Figure 1: A visual depiction of multiple memory protection domains within a single shared address space.

dresses. The address space can be virtual or physical—protection domains are independent from how virtual memory translation is done (if it is done at all). A protection domain can contain many threads, and every thread is associated with exactly one protection domain at any one point in its execution. Protection domains that want to share data with each other must share at least a portion of their address space. The color in each box represents the permissions that each protection domain has to access the region of memory. An ideal protection system would allow each protection domain to have a unique view of memory with permissions set on arbitrary-sized memory regions.

The system we present in this paper implements this ideal protection system. We call this *Mondrian memory protection* (MMP) because it allows the grid in Figure 1 to be painted with any pattern of access permissions, occasionally resembling works by the eponymous early twentieth century artist. Our design includes all of the flexibility and high-performance protected memory sharing of a segmented architecture, with the simplicity and efficiency of linear addressing. The design is completely compatible with existing ISAs, and can easily support conventional operating system protection semantics.

To reduce the space and run-time overheads of providing fine-grained protection, MMP uses a highly-compressed permissions table structure and two levels of hardware permissions caching. MMP overheads are less than 9% even when the system is used aggressively to provide separate protection for every object in a program. We believe the increase in design robustness and the reduction in application design complexity will justify these small run-time overheads. In some cases, the new application structure enabled by fine-grained protection will improve performance by eliminating cross-context function calls and data copying. We demonstrate this by saving 52% of the memory traffic in our zero-copy networking implementation (see Section 5.3). The networking example also illustrates a fine-grain segment translation scheme which builds upon the base MMP data structures to provide a facility to present data at different addresses in different protection domains. The MMP design also has the desirable property that the overhead is only incurred when fine-grain protection is used, with less than 1% overhead when emulating conventional coarse-grained protection.

The rest of the paper is structured as follows. In Section 2 we give a motivating example and discuss our requirement for memory system protection. Then we present the hardware and software components of the MMP design in Section 3. We quantitatively measure the overheads of our design in our implementation model in Section 4. We discuss translation in Section 5 and describe its use in zero-copy networking. We include a discussion of uses for fine-grained protection and sharing in Section 6, and a discussion of related work in Section 7. We conclude in Section 8.

2. EXAMPLE AND REQUIREMENTS

We provide a brief example to motivate the need for the MMP system. More examples are discussed in Section 6. Consider a network stack where when a packet arrives, the network card uses DMA to place a packet into a buffer provided to it by the kernel driver. Instead of the kernel copying the network payload data to a user supplied buffer as is normally done, the kernel makes the packet headers inaccessible and the packet data read-only and passes a pointer to the user, saving the cost of a copy.

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

- `different`: Different protection domains can have different permissions on the same memory region.
- `small`: Sharing granularity can be smaller than a page. Putting every network packet on its own page is wasteful of memory if packets are small. Worse, to give the header separate permissions from the payload would require copying them to separate pages in a page-based system (unless the payload starts at a page boundary).
- `revoke`: A protection domain owns regions of memory and is allowed to specify the permissions that other domains see for that memory. This includes the ability to revoke permissions.

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

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

Page-group systems [16], such as HP-PA RISC and PowerPC, define protection domains by which page-groups (collections of memory pages) are accessible. Every domain that has access to a page-group sees the same permissions for all pages in the group, violating the `different` requirement. They also violate the `small` requirement because they work at the coarse granularity of a page or multiple pages. Domain-page systems [16] are similar to our design in that they have an explicit domain identifier, and each domain can specify a permissions value for each page. They fail to meet the `small` requirement because permissions are managed at page granularity.

Capability systems [10, 18] are an extension of segmented architectures where a capability is a special pointer that contains both location and protection information for a segment. Although designed for protected sharing, these fail the `different` requirement for the common case of shared data structures that contain pointers. Threads sharing the data structure use its pointers (capabilities) and therefore see the same permissions for objects accessed via the shared structure. Many capability systems fail to meet the `revoke` requirement because revocation can require an exhaustive sweep of the memory in a protection domain [7]. Some capability systems meet the `different` and `revoke` requirements by performing an indirect lookup on each capability use [13, 29], which adds considerable run-time overhead.

Large sparse address spaces provide an opportunity for probabilistic protection [35], but this strategy violates the `revoke` and `different` requirement.

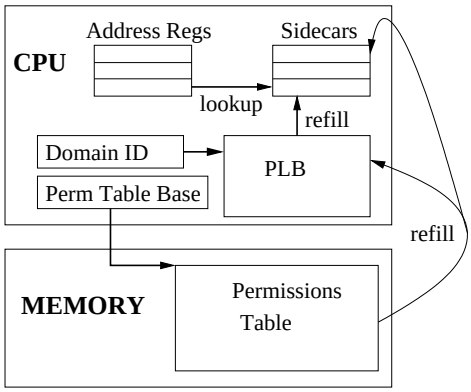


Figure 2: The major components of the Mondrian memory protection system. On a memory reference, the processor checks permissions in the address register sidecar. If the reference is out of range of the sidecar information, or the sidecar is not valid, it attempts to reload the sidecar from the PLB. If the PLB does not have the permissions information, either hardware or software walks the permissions table which resides in memory. The matching entry from the permissions table is cached in the PLB and is used to reload the address register sidecar with a new segment descriptor

3. MMP DESIGN

The major challenge in a MMP system is reducing the space and run-time overheads. In the following, we describe our initial exploration of this design space and our trial implementation. The implementation and results are for a 32-bit address space, but MMP can be readily extended to 64-bit addresses as discussed in Section 3.9.

3.1 MMP Features

MMP provides multiple protection domains within a single address space (physical or virtual). Addressing is linear and is compatible with existing binaries for current ISAs. A privileged supervisor protection domain is available which provides an API to modify protection information. A user thread can change permissions for a range of addresses, a *user segment*, by specifying the base word address, the length in words, and the desired permission value. Changing memory protections only incurs the cost of an inter-protection domain call (Section 3.8), not a full system call.

In all the designs discussed in this section, we provide two bits of protection information per word, as shown in Table 1. MMP can be easily modified to support more permission bits or different permission types.

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

Table 1: Example permission values and their meaning.

Every allocated region of memory is owned by a protection domain, and this association is maintained by the supervisor. To support the construction of protected subsystems, we allow the owner of a region to export protected views of this region to other protection domains.

3.2 MMP System Structure

Figure 2 shows the overall structure of an MMP system. The CPU contains a hardware control register which holds the protection domain ID (PD-ID [16]) of the currently running thread. Each domain has a *permissions table*, stored in privileged memory, which specifies the permission that domain has for each address in the address space. This table is similar to the permissions part of a page table, but permissions are kept for individual words in an MMP system. Another CPU control register holds the base address of the active domain's permissions table.

The MMP protection table represents each user segment using one or more *table segments*, where a table segment is a convenient unit for the table representation. We use the term *block* to mean an address range that is naturally aligned and whose size is a power of two. In some MMP variants, all table segments are blocks.

Every memory access must be checked to see if the domain has appropriate access permissions. A *permissions lookaside buffer (PLB)* caches entries from the permissions table to avoid long walks through the memory resident table. As with a conventional TLB miss, a PLB miss can use hardware or software to search the permission tables. To further improve performance, we also add a *sidecar register* for every architectural address register in the machine (in machines that have unified address and data registers, a sidecar would be needed for every integer register). The sidecar caches the last table segment accessed through this address register. As discussed below, the information stored in the sidecar can map a wider address range than the index address range of the PLB entry from which it was fetched, avoiding both PLB lookups and PLB misses while a pointer moves within a table segment. The information retrieved from the tables on a PLB miss is written to both the register sidecar and the PLB.

The next two subsections discuss alternative layouts of the entries in the permissions tables. In choosing a format of the permissions table we must balance space overhead, access time overhead, PLB utilization, and the time to modify the tables when permissions change.

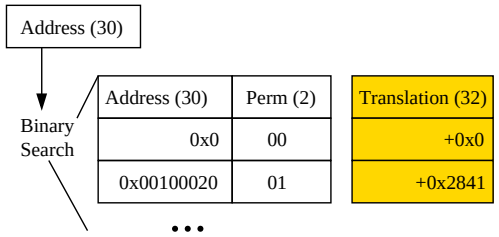


Figure 3: A sorted segment table (SST). Entries are kept in sorted order and binary searched on lookup. The shaded part contains optional translation information.

3.3 Sorted Segment Table

A simple design for the permissions table is just a linear array of segments ordered by segment start address. Segments can be any number of words in length and start on any word boundary, but cannot overlap. Figure 3 shows the layout of the sorted segment table (SST). Each entry is four bytes wide, and includes a 30-bit start address (which is word aligned, so only 30 bits are needed) and a 2-bit permissions field (the shaded part is optional and will be discussed in Section 5). The start address of the next segment implicitly encodes the end of the current segment, so segments with no permissions are used to encode gaps and to terminate the list. On a PLB miss, binary search is used to locate the segment containing the demand address. The SST is a compact way of describing the segment structure, especially when the number of segments is

small, but it can take many steps to locate a segment when the number of segments is large. Because the entries are contiguous, they must be copied when a new entry is inserted. Furthermore, the SST table can only be shared between domains in its entirety, i.e., two domains have to have identical permissions maps.

3.4 Multi-level Permissions Table

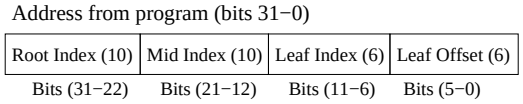


Figure 4: How an address indexes the multi-level permissions table (MLPT).

An alternative design is a multi-level permissions table (MLPT). The MLPT is organized like a conventional forward mapped page table, but with an additional level. Figure 4 shows which bits of the address are used to index the table, and Figure 5 shows the MLPT lookup algorithm. Entries are 32-bits wide. The root table has 1024 entries, each of which maps a 4 MB block. Entries in the mid-level table map 4 KB blocks. The leaf level tables have 64 entries which each provide individual permissions for 16 four-byte words. The supervisor can reduce MLPT space usage by sharing lower level tables across different protection domains when they share the same permissions map.

We next examine different formats for the entries in the MLPT.

3.4.1 Permission Vector Entries

A simple format for an MLPT entry is a vector of permission values, where each leaf entry has 16 two-bit values indicating the permissions for each of 16 words, as shown in Figure 6. User segments are represented with the tuple < base addr, length, permissions>. Addresses and lengths are given in bytes unless otherwise noted. The user segment `<0xFFC, 0x50, RW>` is broken up into three permission vectors, the latter two of which are shown in the figure. We say an address range *owns* a permissions table entry if looking up any address in the range finds that entry. For example, in Figure 6, `0x1000`–`0x103F` owns the first permission vector entry shown.

Upper level MLPT entries could simply be pointers to lower level tables, but to reduce space and run-time overhead for large user segments, we allow an upper level entry to hold either a pointer to the next level table or a permissions vector for sub-blocks (Figure 7). Permission vector entries in the upper levels contain only eight sub-blocks because the upper bit is used to indicate whether the entry is a pointer or a permissions vector. For example, each mid-level permissions vector entry can represent individual permissions for the eight 512 B blocks within the 4 KB block mapped by this entry.

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

Figure 5: Pseudo-code for the MLPT lookup algorithm. The table is indexed with an address and returns a permissions table entry, which is cached in the PLB. The base of the root table is held in a dedicated CPU register. The implementation of `is_tbl_ptr` depends on the encoding of the permission entries.

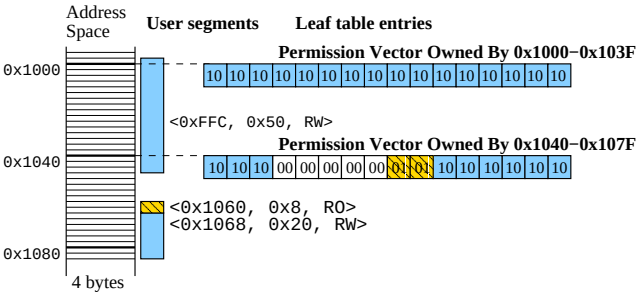
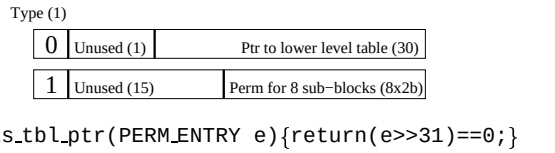


Figure 6: A MLPT entry consisting of a permissions vector. User segments are broken up into individual word permissions.




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

Figure 7: The bit allocation for upper level entries in the permissions vector MLPT, and the implementation of the function used in MLPT_lookup.

3.4.2 Mini-SST entries

Although permission vectors are a simple format for MLPT entries, they do not take advantage of the fact that most user segments are longer than a single word. Also, the upper level entries are inefficient at representing the common case of non-aligned, non-power-of-two sized user segments.

The sorted segment table demonstrated a more compact encoding for abutting segments—only base and permissions are needed because the length of one segment is implicit in the base of the next. A *mini-SST* entry uses the same technique to increase the encoding density of an individual MLPT entry.

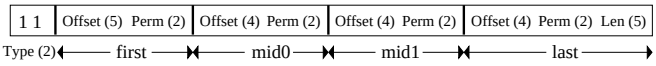


Figure 8: The bit allocation for a mini-SST permission table entry.

Figure 8 shows the bit encoding for a mini-SST entry which can represent up to four table segments crossing the address range of an entry. As with the SST, start offsets and permissions are given for each segment, allowing length (for the first three entries) to be implicit in the starting offset of the next segment. The mini-SST was broken up into four segments because experiments showed that the size of heap allocated objects was usually greater than 16 bytes.

Mini-SST entries encode permissions for a larger region of memory than just the 16 words (or 16 sub-blocks at the upper table levels) that own it. The `first` segment has an offset which represent its start point as the number of sub-blocks (0–31) *before* the base address of the entry's owning range. Segments `mid0` and `mid1` must begin and end within this entry's 16 sub-blocks. The `last` segment can start at any sub-block in the entry except the first (a

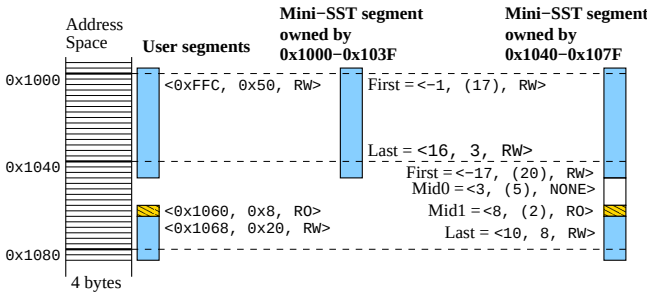


Figure 9: An example of segment representation for mini-SST entries.

zero offset means the `last` segment starts at the end address of the entry) and it has an explicit length that extends up to 31 sub-blocks from the end of the entry's owning range. The largest span for an entry is 79 sub-blocks (31 before, 16 in, 32 after).

The example in Figure 6 illustrates the potential benefit of storing information for words beyond the owning address range. If the entry owned by `0x1000`–`0x103F` could provide permissions information for memory at `0x1040` then we might not have to load the entry owned by `0x1040`.

Figure 9 shows a small example of mini-SST entry use. Segments within an SST entry are labelled using a < base, length, permission > tuple. Lengths shown in parentheses are represented implicitly as a difference in the base offsets of neighboring table segments. The entry owned by `0x1000-0x103F` has segment information going back to `0xFFC`, and going forward to `0x104C`. Because of the internal representation limits of the mini-SST format, the user segment mapped by the entry at address range `0x1000-0x103F` has been split across the first and last mini-SST table segments.

Mini-SST entries can contain overlapping address ranges, which complicates table updates. When the entry owned by one range is changed, any other entries which overlap with that range might also need updating. For example, if we free part of the user segment starting at `0xFFC` by protecting a segment as `<0x1040, 0xC, NONE>`, we need to read and write the entries for both `0x1000`–`0x103F` and `0x1040`–`0x107F` even though the segment being written does not overlap the address range `0x1000`–`0x103F`. All entries overlapping the modified user segment must also be flushed from the PLB to preserve consistency.

We can design an efficient MLPT using mini-SST entries as our primary entry type. The mini-SST format reserves the top two bits for an entry type tag, Table 2 shows the four possible types of entry. The upper tables can contain pointers to lower level tables. Any level can have a mini-SST entry. Any level can contain a pointer to a vector of 16 permissions. This is necessary because mini-SST entries can only represent up to four abutting segments. If a region contains more than four abutting segments, we represent the permissions using a permission vector held in a separate word of storage, and pointed to by the entry. Finally, we have a pointer to a record that has a mini-SST entry and additional information. We use this extended record to implement translation as discussed in Section 5.

3.5 Protection Lookaside Buffer

The protection lookaside buffer (PLB) caches protection table entries in the same way as a TLB caches page table entries. The PLB hardware uses a conventional ternary content addressable memory (CAM) structure to hold address tags that have a varying number of significant bits (as with variable page size TLBs [15]). The PLB tags have to be somewhat wider than a TLB as they support finer-grain addressing (26 tag bits for our example design). Entries are also tagged with protection domain identifiers (PD-IDs).

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

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

Table 2: The different types of MLPT entries, and the implementation of the function used in MLPT_lookup. *Type* is the type code. Leaf tables do not have type `00` pointers.

The ternary tags stored in the PLB entry can contain additional low-order "don't care" address bits to allow the tag to match addresses beyond the owning address range. For example, the tag `0x10XX`, where `XX` are don't care bits, will match any address from `0x1000`–`0x10FF`. On a PLB refill, the tag is set to match on addresses within the largest naturally aligned power-of-two sized block for which the entry has complete permissions information. Referring to the example in Figure 9, a reference to `0x1000` will pull in the entry for the block `0x1000`–`0x103F` and the PLB tag will match any address in that range. A reference to `0x1040` will bring in the entry for the block `0x1040`–`0x107F`, but this entry can be stored with a tag that matches the range `0x1000`–`0x107F` because it has complete information for that naturally aligned power-of-two sized block. This technique increases effective PLB capacity by allowing a single PLB entry to cache permissions for a larger range of addresses.

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

3.6 Sidecar Registers

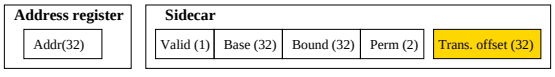


Figure 10: The layout of an address register with sidecar. The shaded portion is optional translation information.

Each address register in the machine has an associated sidecar register which holds information for one table segment as depicted in Figure 10. The program counter also has its own sidecar used for instruction fetches. Sidecar registers are an optional component of the design, but they help reduce traffic to the fully-associative PLB.

On a PLB miss, the demand address from the processor is looked up in the permissions table, and the permissions table entry that is returned is entered into the PLB. The table segment that contains the demand address is also loaded into the sidecar for the address register that was used to calculate the effective address of the memory load or store. All fields of the table segment descriptor are blown up to maximum length in the address sidecar to facilitate fast checking of base and bounds. For each subsequent load or store, the effective address is compared against the base and bounds. If the address lies within the range, the sidecar permissions value is used to check the access. If the range check fails or the sidecar is invalid, the PLB is searched for the correct permissions information. The PLB in turn might miss, causing a fill from the permissions table in memory.

Sidecars also increase the permissions hit rate by caching an entire table segment. The PLB can often only index part of the permission table entry because its index range must be a naturally aligned power-of-two sized block. For example, in Figure 9 a reference to `0x1040` will load the segment `<0xFFC, 0x50, RW>` into the register sidecar. If that register is used to access location `0xFFC` we will have a permissions check hit from the sidecar. Sending `0xFFC` to the PLB will result in a permissions check miss because it only indexes the range `0x1000`–`0x107F`.

To guarantee consistency, all sidecars are invalidated when any protections are changed. Sidecars are also invalidated on protection domain switches. The sidecars can be refilled rapidly from the PLB. Permissions tables have the same coherence issues as page tables in a multi-processor system. If they are modified, any processor which might be caching the data must be notified so it can invalidate its sidecar registers and invalidate the necessary section of the PLB.

Register sidecar information is like a capability in that it has protection and range information, but it is not managed like a capability because it is ephemeral and not user visible. Sidecars are similar to the resolved address registers in the IBM System/38 [13], where an address such as the base of an array would be translated, cached and then reused to access successive array elements.

3.7 Processor Pipeline Implementation

MMP requires some modifications to a processor pipeline, but the permissions check only needs to occur before the commit point in the pipeline, and so should not impact processor cycle time. A permissions fault is treated in the same way as any other address fault. In an in-order processor, the check is performed before write back, and in an out-of-order processor the check is performed before instruction retirement. A processor can speculatively use load data before permission is granted. Similarly, store data lives in the speculative store buffer until permission is granted to commit the store.

The sidecar registers can be physically located by the load/store unit and only need as many read ports as the number of simultaneous load and store instructions supported. For both in-order and out-of-order processors, the architectural register number is used to index the sidecar register file. The sidecar registers in an out-of-order pipeline may be speculatively refilled with the wrong data, but the segment descriptor will always be for a valid table segment. An incorrect speculation might bring in the wrong table segment, but this can only cause a sidecar miss to the PLB, not a protection breach.

The main performance impact of permissions checking is the additional additional memory traffic caused by table lookups. This is quantified below in Section 4.

3.8 Protected calls

The memory protection structures of the permissions table and PLB are sufficient to implement call gates [26]. Call gates are generalizations of system calls, and provide an efficient mechanism for mutually distrustful protection domains to safely call each other's services.

A subsystem exports a limited number of code entry points to client domains. Calls to these entry points cause a switch in protection domain to that of the subsystem. There are a number of ways to implement these protected entry points. The simplest is to remove all permissions on the entry points so that a call will trap into the supervisor. Subsystems register their entry points with the supervisor, so execution can be restarted in the exporting domain at the requested entry point. Alternatively, hardware can be used to accelerate the domain switch by encoding the protection domain for a code entry point within the permissions table entry.

The call gate has a minimum of semantics. For instance the exporting domain needs to establish its own stack if it needs one. Parameters are passed in registers. More elaborate data structures are passed using a very simplified form of marshalling which consists of the caller traversing the data structure and granting appropriate permission to the provider domain. No data copying is needed, only the permissions structures are affected. If two domains call each other frequently, they can copy arguments into buffers which are properly exported.

Hardware-supported call gates make cross-domain calls efficient by removing the demultiplexing overhead of system calls. This allows supervisor services to be implemented with a minimum of checking. `Malloc` and `free` could be implemented efficiently by the supervisor since it need only check the length on a `malloc`, and it need only verify the pointer on a `free`. There is no transfer of data. To further speed execution within the supervisor domain, we assume a small number of wired entries are reserved exclusively for supervisor use (4 in our implementation). The supervisor can keep protection information for its text, stack, and data in these entries so they do not need to be faulted in on every supervisor call.

3.9 Extension to 64 bits

Although this paper only describes an MMP design for a 32-bit address space, we believe the design can be extended to 64 bits in a straightforward way, using many of the same techniques that were used to extend page tables to a wider address space. Using a forward-mapped scheme, we would have five levels of table lookup, where the top 3 level tables have 4K entries, and the last two levels have 2K entries. To make lookups faster, we can hash the top 42 bits of the address and index into an address hash table which has either permission entries or pointers to the same lowest two level tables used by the five table lookup path. The hash table is updated whenever a lookup fails and it is updated with the entry retrieved from searching the five level tables from the root. The space consumption for this strategy will be larger than the 32-bit case, but we believe the time consumption could be tuned to be close to the 32-bit case.

4. EVALUATION

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

Data placement is governed by the executable program layout and by the operation of the heap allocator. We evaluated both C

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

Table 3: The reference behavior of benchmarks. The *Refs* column is total number of loads and stores in millions. The *Segments* column is the number of segments written to the table (which is twice the number of calls to `malloc` since each call effectively creates two segments). The next column is the average number of memory references between updates to the permissions table. *Cs* is the number of segments when running with coarse-grained protection. These come from initial program segments, from calls to `brk` (extending the heap), or from extending the stack.

and Java programs. C programs were compiled with `gcc` version `egcs-1.0.3a` for a 32-bit MIPS target using -O3 optimization and static linking to generate an ELF binary. The malloc from the `newlib` library was used. The linker and malloc libraries were used unmodified. The results would be significantly better if the linker was modified to align and pad program sections, and if malloc was modified to try to align addresses and to place its internal management state away from the returned memory. The Java programs were compiled for a MIPS target using MIT's FLEX Java-to-native compiler [25]. FLEX's output was also linked with the `newlib` malloc library. The garbage collector in FLEX was disabled for all of our runs to put a heavier load on the memory system.

The biggest challenge in evaluating MMP is trying to predict how programmers would take advantage of word-granularity protection. In this evaluation, we considered two extreme cases. In the first case, we assumed light use of the protection facilities. Programs were run with the standard protection regions for Unix processes: read-only program text, read-only data, read-write data, and stack. This level of protection is what is provided by most current virtual memory operating systems. In the second case, we assume that every object allocated by malloc is in a separate user segment and that the surrounding words are inaccessible because they hold malloc internal state.

To gather data on how programs access data, we chose a mix of benchmarks that were both memory reference and memory allocation intensive. Table 3 lists the benchmarks used and their reference properties. Benchmark names prefixed with a "j-" are Java programs. Benchmarks `crafty`, `gcc`, `twolf` and `vpr` are from SPEC 2000, and `vortex` is from SPEC 95. The `_tr` suffix indicates the training input, and `_test` suffix indicates the test input. Names prefixed "o-" are from the Olden [6] benchmark suite. Names prefixed with "m-" are from the Mediabench benchmark suite. Table 3 includes the number of memory references per table update. The permissions table is only updated on `malloc`, `realloc`, and `free` calls, and the results show a wide variation in how frequently objects are created and deleted.

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

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

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

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

4.1 Coarse-Grained Protection Results

Table 4 shows the space and time overhead results for the coarse-grained protection model. We only present the results for the MLPT with mini-SST entries and a 60-entry PLB. We contrast the overheads of the permissions table with a model of a page table and TLB which would provide this same kind of protection in a modern computer system. The overheads are small in both space and time for both systems. The MLPT space overhead is bigger than the page table overhead, but it is less than 0.7% for all of the benchmarks. MLPT uses additional space because it must create a few leaf level tables to accomodate segments whose start or end addresses are not divisible by 256 B. If the program segments were aligned, and grew in aligned quantities, the MLPT and page table would consume the same space.

The MLPT adds fewer than 0.6% extra memory references, and requires fewer table accesses than the page table for every benchmark except Mediabench's `mpeg2`. The `mpeg2` run is so short that writes to the permission table make up a large part of the table accesses. The advantage of the MLPT is the reach of its mid-level mini-SST entries. These entries are owned by 4 KB of address space, but they can contain information for a 20 KB region. A conventional page table entry only has information for the 4 KB range

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

Table 4: The extra memory references *X-ref* and extra storage space *Space* required for a mini-SST permissions table and 60 entry PLB used to protect coarse-grain program regions. We compare to a traditional page table with a 60 entry TLB. The *l/k* column gives the average number of loads required for a table lookup, which is a measure of how much the mid level entries are used for permission information.

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

Table 6: Measurements of miss rates for a MLPT with mini-SST entries and a 60 entry PLB. *SCar* is the sidecar miss rate. *PLB* is the global PLB miss rate (PLB misses/total references). *SCar Elim* is the number of references to the permissions table that were eliminated by the use of sidecar registers for the fine-grained protection workload. For coarse-grained protection, the PLB miss rates were close to zero on all benchmarks and so are not shown here.

that owns it. For instance, `compress`, a benchmark known to have poor TLB performance, mallocs a 134 KB hash table which is accessed uniformly. This table requires 33 TLB entries to map, but would only require 8 entries in the worst case for the PLB. The number of loads per lookup is close to 2 indicating that mid level entries are heavily used.

We also simulated an SST with a 60-entry PLB. This performs much better that either of the previous schemes, with both time and space overheads below 0.01% on all benchmarks. The ability of the SST table segments to represent large regions results in extremely low PLB miss rates. Because there are so few coarse grain segments, the lookup and table update overhead is small.

These results show that the overhead for MMP word-level protection is very low when it is not being used.

4.2 Fine-Grained Protection Results

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

Table 5 shows the results for the fine-grain protection workloads. While the SST organization performs well for some programs, its time and space overhead balloons on other programs. For `o-health` the space overhead reaches 44%. The binary search lookup has a heavy, but variable, time cost, which can more than double the number of memory references. For `j-jack`, it averages 20.8 loads per table lookup, but for `mpeg2` it is only 4.8. Because SST must copy half the table on an update on average, updates also cause significant additional memory traffic. But SST does have significantly lower space and time overheads than the MLPT for some applications like `gcc` and `crafty`. The `gcc` code mallocs a moderate number of 4,072 byte regions for use with its own internal memory manager. This odd size means the MLPT must use leaf tables which have limited reach in the PLB, while the SST represents these segments in their entirety. We are investigating adaptive policies that would switch between SST and MLPT as the number of segments increase.

All MLPT organizations take almost exactly the same space and so their space overhead is reported together in one column. The space overhead for the MLPT is less than 9% for all permission entry types. The mini-SST format can require a little more space than the permission vector format when programs use many small segments that cannot be represented in the mini-SST format. Five of the benchmarks required permission vector escapes, but only two required more than 30 escapes. The `health` benchmark required 4,037 pointers to permissions vectors in the leaf entries, and `j-jess` 332. Although is not likely to represent real program behavior [36], `health` provides a stress test for our system because it allocates many small segments.

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

The mini-SST organization is clearly superior to the permission vector format (compare columns **vec 60 PLB** to **mSST 60 PLB**). Every benchmark performs better and the highest overhead (`vpr`) is more than halved, dropping from 19.4% to 7.5%. Lookups dominate the additional memory accesses, as would be expected. `jpeg` and `mpeg` from mediabench are small programs that don't run for very long so updating the tables is a noticable fraction of table memory references for these benchmarks. `j-jack` has high update overhead because it performs many small allocations with little activity in between (from Table 3 it does less than 500 memory reference in between table updates). When we increase the number of available PLB entries to 124 (column **mSST 124 PLB**), the worst case memory reference overhead drops to 6.3%, with some benchmarks, like `vpr`, benefiting greatly from the increased reach of the PLB.

4.3 Memory Hierarchy Performance

Table 6 shows the performance of the permissions caching hi-

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

Table 5: Comparison of time and space overheads with inaccessible words before and after every malloced region. The *Space* column is the size of the permissions table as a percentage of the application's active memory. The last three organizations are all MLPT and all occupy about the same space. The *X-Ref* column is the number of permissions table memory accesses as a percentage of the application's memory references. The *upd* colum indicates the percentage of table memory accesses that were performed during table update. The remainder of the references are made during table lookup. The *ld/lk* column gives the average number of loads required for a table lookup.

erarchy including the sidecar miss rate and the PLB global miss rate for the fine-grained protection workload. The sidecar registers normally capture 80–90% of all address accesses, while the PLB captures over 97% in all cases.

We also show the percentage reduction in references to the permissions tables as a result of using sidecar registers. The principal motivation for using sidecars is to reduce traffic to the PLB, but there is a significant performance gain also (more than 10% for two benchmarks) because some sidecar hits would be PLB misses as explained in Section 3.6.

As another indirect measure of performance impact, we measured the increase in miss rate caused by the additional permissions table accesses. The results for a typical L1 cache (16 KB) and a typical L2 cache (1 MB) are shown in Figure 7. Both caches are 4-way set associative. For the L1 cache, at most an additional 0.25% was added to the miss rate, and for the L2 cache, at most 0.14% was added to the global miss rate but most apps experienced no difference in L2 miss rates.

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

Table 7: *App* is the cache miss rate of the application benchmark, while *MMP* is the combined cache miss rate for the references of the benchmark and the MMP protection structures. Δ is their difference. A MLPT was used with mini-SST entries and a 60 entry PLB. This table holds results from two experiments, differing only in cache size—16 KB and 1 MB. The cache was a 4-way set-associative with 32-byte lines. -0.00 means the miss rate decreased slightly. Reference streams were simulated for a maximum of 2 billion references.

5. SEGMENT TRANSLATION

The MMP table structures are effective at associating permissions with memory addresses. Other information can also be associated with addresses and held in the table segment descriptors. We can make a segment of memory appear to reside in a different address range by storing a translation offset in the table segment descriptor. The translation offset is added in to every address calculation within the table segment's range.

Figure 11 shows an example of how this facility might be used. Addresses in the range `0x1000-0x12FF` actually refer to memory stored at the two different address regions `0x80002000`–`0x800021FF` and `0x80002800`–`0x800028FF`. This is implemented by the creation of two segments that have translation information, i.e., `<0x1000, 0x200, RO, +0x80001000>`, and `<0x1200, 0x100, RO, +0x80001600>`. The final segment field holds the translation offset.

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

Figure 11 shows how translation can be used to implement zero-copy networking with a standard `read` system call interface. The client domain passes a buffer to the kernel via `read`. The kernel becomes the owner of the buffer, and it remaps the packet payloads into the buffer without copying them. When the user references the buffer (e.g., `0x1000`), it is reading data from `0x80002000` which is where the packet payload resides.

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

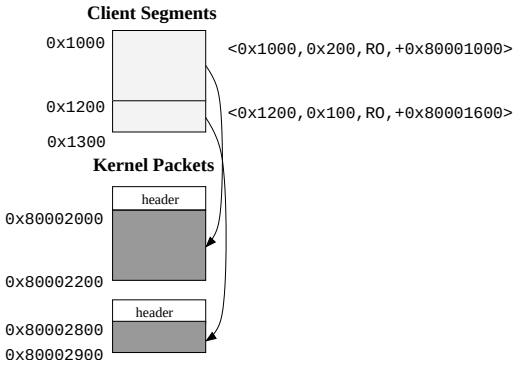


Figure 11: Using memory protection and segment translation to implement zero-copy networking. The network interface card DMAs packets into the kernel. The kernel exports the packets to an untrusted client by creating segments for the payload of the packets. Segment translation is used to present the illusion to the client that the packet payloads are contiguous in memory at `0x1000-0x12FF`.

recursive, a translated segment cannot be the target of other translations.

5.1 Byte Granularity Translation

To allow packet payloads to consist of any number of bytes, segment translation must be done at the byte level. Byte level translation creates two issues.

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

The second issue is a little more complex. Returning to the example in Figure 11, consider the case where the first packet has one fewer byte of data payload: `0x1FF` bytes instead of `0x200`. We can almost represent this situation with the segments `<0x1000, 0x1FF, RO, +0x80001000>` and `<0x11FF, 0x101, RO +0x80001601>`, but the length of our segments and their base address must be word aligned, they can not be byte aligned. The problem is with the word at address `0x11FC`. The first three bytes need to come from the first segment, and the last byte needs to come from the second segment.

We call a word that spans segment translation boundaries a *seamed word*. Seamed words must be represented in the permissions table. To simplify the representation, they are defined to be single word segments that must occur on the first word of two adjacent segments, e.g., the word at address `0x11FC` in our example. We then only need to represent that two adjacent segments have a seam and how many bytes from the first segment are used. The remaining bytes are taken from the second segment.

Figure 12 shows the record used to represent seamed words and translation information in the mini-SST format. The record is six words long and is pointed to by a table entry which is a type `10` pointer (see Table 2). There are 32 bits of translation for each segment.

The upper two bits, used for type information when the mini-SST format is used for table entries, are reallocated in this record to indicate the location of seamed words. We can use these bits

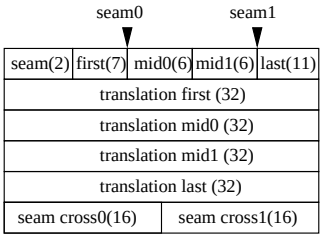


Figure 12: The format for a record with a mini-SST entry and translation information.

because we already know the type of the entry once we reach it. The arrow heads indicate where seamed words are allowed to occur. The bits are independent and if the first bit (`seam0`) is set, a seam is between table segments `first` and `mid0`. If the second bit is set(`seam1`), a seam is between `mid1` and `last`. We divide the last word of the record into two 16-bit fields, which each represent the byte cross-over point for the corresponding seamed entry. In our example, the cross-over point is 3 bytes because `0x11FC`–`0x11FE` come from the first segment and `0x11FF` comes from the second segment.

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

5.2 Translation Hardware Implementation

The translation offset sits in the address sidecar register (Figure 10) and must be added in to every memory address calculation. This will increase the typical two operand add used for address arithmetic to a three operand add. The additional 3:2 carry-save adder will add a few gate delays to memory access latency.

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

Segment translation does not cause cache hardware aliasing problems, because translation occurs before the access is sent to the cache and memory system. There can be a software pointer aliasing problem if software assumes that only numerically equal pointers point to the same data. Since all memory meta-data is changed via supervisor calls, the supervisor can enforce policies that mitigate the negative effects of software pointer aliasing. One policy would be that, since a domain must own both the translated segment and its image, the domain can only export the segment, and not the image. This prevents other domains from seeing the translation and becoming confused, but would support applications like zero-copy networking.

5.3 Zero-copy Networking

There are many proposals in the literature for zero-copy networking [9, 23, 30]. Most are successful at eliminating extra copies in the kernel. The hardest implementation issue is eliminating the copy between the kernel and the user. Systems like IOLite [23] change the user/kernel interface and programming model to pass around collections of pointers. The user is aware that her data is split into various memory regions which complicates program-

ming. Another approach has user handlers manage the copy from the network interface directly [20]. Direct access to the network interface requires special hardware, does not interact well with multiprogramming and demand paging, and results in the entire packet, not just the payload, being transfered to user space. A final approach [9] uses page remapping, which can be implemented under the standard `read` system call. The implementation in [9] is limited to the hardware page granularity, and so only applicable within large packets (the largest standard Ethernet packet is less than 1600 bytes).

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

The kernel buffers packets as they arrive on a TCP connection. It then maps the payload from these packets into contiguous segments (provided by `read`) which the user can then access (see Figure 11). Permissions are only given for access to the data payload so the network stack is isolated from a malicious or buggy user.

5.3.1 Evaluation

We recorded a web client receiving 500 KB of packets and simulated the action of a kernel driver which accepts the packets into kernel memory and then translates the packet payload segments into a contiguous segment which is exported to the client. The client then streams through the entire payload. In this scenario, the kernel reads the packet headers, and writes the permissions tables to establish the translation information. The client reads the data, causing the system to read the translation and permissions data from the protection table.

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

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

6. OTHER USES FOR FINE-GRAINED PROTECTION AND TRANSLATION

We believe that fine-grained protection offers exciting opportunities for application developers. Appel [3] surveys some applications that make use of page-based virtual memory. Many of these same ideas could perform better with finer grain protection and with cheap inter-protection domain calls.

Fine-grained protection can provide support for fast memory bounds checking. Buffer overruns in unsafe languages are a common source of security holes [32]. MMP could catch a program's attempt to jump into writable data. It could also catch the program trying to write off the end of a piece of memory. Bounds checking is useful for program debugging and if implemented by MMP would be available to the kernel.

A related functionality, data watchpoints [33], can be easily implemented with our fine-grained protection. A data watchpoint generates a trap when a given word in memory is modified. Some processors support a handful of watched memory locations [15, 14], but our fine-grained protection scales to thousands of individually protected words.

Generational garbage collectors [19] need to be notified when older objects are updated to point to younger ones. Checking this in software is time consuming. With MMP, we can write protect older objects and signal whenever an update causes a young object to be referenced by an old object.

Compilers for unsafe languages like C are often unable to apply compelling optimizations because of their inability to prove something about the memory reference behavior of the program. The following loop illustrates the point.

```
void foo(int* a, int B[]) {
   for(int i = 0; i < N; ++i) {
      *a += B[i]; }}
```

The compiler can not register allocate `*a` if it can not prove that `a` and `B` are not aliases. With fine-grained protection, the compiler can write-protect `B` outside the loop and then accumulate `*a` in a register. Fix up code is needed in case `B` is written.

Flexible sub-page protection enables distributed shared memory systems like Shasta [27] and its predecessor [28]. Shasta found significant benefit from configurable line sizes, but since these line sizes did not map to virtual address pages it performed access checks in software. While the authors of Shasta used impressive compiler techniques to reduce the cost of these software access checks, our fine-grained protection would reduce this cost further. In addition, fine-grained protection can be used to perform object-level distributed caching, rather than standard block based caching which is susceptible to false sharing.

6.1 Combining fine-grained protection and translation

When fine-grained protection is combined with byte level translation, we discover additional opportunities for implementing system services. We explored one application in detail, zero-copy networking, in Section 5.3.

A persistent problem for supporting large numbers of user threads is the space occupied by each thread's stack [11]. Each thread needs enough stack to operate, but reserving too much stack space wastes memory. With paged virtual memory, stack must be allocated in page sized chunks. This strategy requires a lot of physical memory to support many threads, even though most threads don't need a page worth of stack space. With MMP segment translation, the kernel can start a thread and only translate a very small part of its stack (e.g., 128 bytes). If the thread uses more stack than this, the kernel can translate the next region of the stack to a segment non-contiguous with the first, so the stack only occupies about as much physical memory as it is using, and that memory does not have to be physically contiguous.

A common data structure that MMP protection and translation could optimize is the mostly-read-only data structure. An example comes from the widely used NS network simulator [22]. Each packet is mostly read-only data. When simulating a wireless network, packets are "broadcast" to nodes which read the read-only data, but also write a small node-specific scratch area in the packet (e.g., to fill in the receive power which is node specific). The current NS simulator supports this data structure by copying the packet for each node. This copying reduces the size of simulations that are possible with a given amount of physical memory, and takes

cycles that could be used for computation. Splitting the packet into read-only and read-write sections and managing them separately is possible, but it complicates a core data structure. By using fine-grain MMP translation, a single read-only payload can be made visible at different addresses within multiple protection domains. Each domain can then have a private read/write region allocated to run contiguous to the read-only view.

7. RELATED WORK

In Section 2, we discussed the problems with page-based virtual memory, segment-based architectures, capabilities, and probabilistic protection address spaces.

Single-address space operating systems (SASOSes) place all processes in a single large address space, so any process can attempt to access any address within the space. Protection is provided by per-process protection domains which specify the access rights for regions of memory. Management of protection information can be separated from paging information [16], though these are often combined. The granularity of protection in single-address space systems is usually a page to match the capabilities of the underlying paging hardware. An advantage of the protection domain approach is that conventional pointers can be used, and permissions can be easily revoked by modifying the per-process permissions tables. *Domain-page systems* [16] can only set permissions at the granularity of a memory page. Page-group systems, such as HP PA-RISC and PowerPC, require that a collection of pages are mapped together and with the same permissions across all domains (though each domain independently might or might not have access at that fixed permission) [16]. MMP builds upon the SASOS protection domain approach but extends it to word granularity. MMP can be considered a *domain-segment* system.

The Apple Newton also had a form of page-group system, where an active process had access to a set of regions (called domains) which had the same access permissions across all processes. The ARM940T is a recent embedded processor that allows the active process to access eight overlapping segments of the global physical address space, but the segments must have power-of-2 size and alignment with a minimum size of 4 KB [4].

Research on SASOSes has concentrated on large virtual address spaces where pointers are immutable names [8, 12]. Another important application area is embedded systems, which often have a single small physical address space. Capabilities are one popular solution for permissions control in SASOSes, but they are ill suited to embedded systems because of the need to reuse physical addresses which requires efficient rights revocation. In contrast, MPP is well suited for use within a single small physical address space embedded system because revocation is straightforward.

There are a range of software techniques for memory protection. Software fault isolation [34] is a general technique that restricts the address range of loads and stores by modifying a program binary. Purify [24] is a software solution for memory bounds checking based on executable rewriting. It has gained wide acceptance, however it can't be used in an OS kernel, or in some embedded development environments since required system services are often not available in these environments and the allocators for these systems tend to have individual, non-standard semantics. Executable rewriting systems degrade performance considerably. Safe langauge techniques [31] degrade performance less, but are only applicable to a given target language. Proof-carrying code [21] is a system where software carries its own proof of safety that just needs to be checked at run-time, but only works for small pieces of code. An MMP system can run arbitrary code for existing ISAs at high speed.

8. CONCLUSION

We have presented and evaluated a proposal for fine-grained memory protection and translation. MMP is compatible with current architectures and ISAs. It supports flexible sharing of data between applications and simplifies the construction of protected subsystems. Compared with previous protection domain approaches, we remove the restriction that permissions are managed at page granularity. Compared to capabilities, we provide a more flexible permissions model, fast rights revocation, and compatibility with current linearly addressed ISAs. Our feasibility study indicates that the space and run-time overhead of providing this fine-grain protection is small and scales with the degree to which fine-grain protection is used. Our zero-copy networking example shows how our new facilities can be used to implement efficient applications.

9. ACKNOWLEDGEMENTS

We would like to thank Frans Kaashoek (who would have preferred we used the Dutch spelling, Mondriaan), Ronny Krashinsky, Chris Batten, John Jannotti, Doug DeCouto and the anonymous reviewers for their comments and Benjie Chen for the NS example. This work is supported by DARPA PAC/C award F30602-00-2-0562, NSF CAREER award CCR-0093354, and a donation from Infineon Technologies.

10. REFERENCES

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

[2] Apache Software Foundation. *mod_perl*, 2002. `http://perl.apache.org/`.

[3] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of ASPLOS-IV*, April 1991.

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

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

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

[7] N. P. Carter, S. W. Keckler, and W. J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of ASPLOS-VI*, pages 319–327, San Jose, California, 1994.

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

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

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

[11] D. Grunwald and R. Neves. Whole-program optimization for time and space efficient threads. In *Proceedings of ASPLOS-VII*, Oct 1996.

[12] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi single-address-space operating system. *Software—Practice and Experience*, 28(9):901–928, 1998.

[13] M. E. Houdek, F. G. Soltis, and R. L. Hoffman. IBM System/38 support for capability-based addressing. In *ISCA*, pages 341–348, 1981.

[14] Intel Corporation. Volume 1: Basic architecture. *Intel*

Architecture Software Developer's Manual, Volume 1: Basic Architecture, 1997.

[15] G. Kane and J. Heinrich. *MIPS RISC Architecture (R2000/R3000)*. Prentice Hall, 1992.

[16] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architectural support for single address space operating systems. In *ASPLOS-V*, pages 175–186, 1992.

[17] B. Lampson. Protection. In *Proc. 5th Princeton Conf. on Information Sciences and Systems*, 1971.

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

[19] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. In *Communications of the ACM 23(6):419–429*, 1983.

[20] K. Mackenzie, J. Kubiatowicz, M. Frank, W. Lee, V. Lee, A. Agarwal, and M. F. Kaashoek. Exploiting two-case delivery for fast protected messaging. In *HPCA*, pages 231–242, 1998.

[21] G. C. Necula. Proof-carrying code. In *POPL '97*, pages 106–119, Paris, Jan. 1997.

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

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

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

[25] M. Rinard and *et al.* The FLEX compiler infrastructure. 1999–2001. `http://www.flex-compiler.lcs.mit.edu`.

[26] J. Saltzer. Protection and the control of information sharing in Multics. *Comm. ACM 17, 7 (July 1974), 388–402*, 1974.

[27] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A low overhead, software-only approach for supporting finegrain shared memory. In *Proceedings of ASPLOS-VII*, Oct 1996.

[28] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *ASPLOS-VI*, 1994.

[29] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *SOSP*, pages 170–185, 1999.

[30] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Symposium on Operating Systems Principles*, pages 303–316, 1995.

[31] T. von Eicken, C.-C. Chang, G. Czajkowski, C. Hawblitzel, D. Hu, and D. Spoonhower. J-kernel: A capability-based operating system for Java. In *Secure Internet Programming*, pages 369–393, 1999.

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

[33] R. Wahbe. Efficient data breakpoints. In *Proceedings of ASPLOS-V*, Oct 1992.

[34] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[35] C. Yarvin, R. Bukowski, and T. Anderson. Anonymous RPC: Low-latency protection in a 64-bit address space. In *USENIX Summer*, pages 175–186, 1993.

[36] C. B. Zilles. Benchmark health considered harmful. *Computer Architecture News*, 29(3), 2001.