Variations of Virtual Memory

CSE 240A Student Presentation
Paul Loriaux
Thursday, January 21, 2010
Mondrian Memory Protection

Emmett Witchel, Josh Cates, and Krste Asanović
MIT Laboratory for Computer Science, Cambridge, MA 02139
{witchel,cates,krste}@lcs.mit.edu
Every user process assigned its own linear address space.

Each address space a single protection domain shared by all threads.

Sharing only possible at page granularity.

Disadvantage 1: Pointer meaningless outside its address context

Disadvantage 2: Transfer of control across protection domains requires expensive context switch.

In other words, sharing is hard and slow.

Compare this to “ideal” VM as imagined years ago.

Every allocated region a “segment” with its own protection information.

However, this has so far proved to be slow and cumbersome. So far...
Enter Mondrian memory protection (MMP)!

Offers fine grained memory protection, with the simplicity and efficiency of today’s linear addressing, with acceptably small run-time overheads.

How? By (A) allowing *different* PDs to have different permissions on the same memory region.

By (B) supporting sharing granularity *smaller* than a page.

By (C) allowing PDs to own regions of memory and grant or *revoke* privileges.

Conventional linear VM systems fail on (A) and (B).

Page-group systems fail on (A) and (B).

Capability-based systems fail mainly on (C), arguably on (A).
1. A Permissions Table, one per PD and stored in privileged memory, specifies the permissions that PD has for every address in the address space.

2. A control register holds the address of the active PD’s permissions table.

3. A PLB caches entries from (1) to reduce memory accesses.

4. A sidecar register, one per address register, caches the last segment accessed by its associated register.

A compressed permissions table reduces space needed for permissions.
How to store permissions, take 1: SST

**Sorted Segment Table**

A linear, sorted array of segments, permitting a binary search on PLB miss.

Segments can be any number of words in length, but cannot overlap.

Each entry in the SST includes a 30-bit start address and a 2-bit permissions field.

Goal: balance (a) space overhead, (b) access time overhead, (c) PLB utilization, and (d) time to modify the tables when permissions change.

Problem: can still take many steps to locate a segment when the number of segments is large.

Problem: Can only be shared between PDs in its entirety.
How to store permissions, take 2: MLPT

**Multi-level Permissions Table**

A multi-level table, sort of like an inode.

1024 entries, each of which maps a 4 MB block, in which each entry maps a 4 KB block, in which each of the 64 entries provides individual permissions for 16 x 4 B words.

---

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

**Code:**

```c
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;
}
```

How are permissions stored in those 4 Byte words?

**Option 1:** Permission Vector Entries

**Option 2:** Mini-SST entries
Well, you’ve got 32 bits, you have 2-bit permissions, so just chop the entry up into 16 2-bit values for indicating the permissions for each of 16 words.

Problem: Do not take advantage of the fact that most user segments are longer than a single word. I.e. not compact.
**Mini-SST Entries**

Two segments (mid0, mid1) encode two different permissions for 16 words.

One segment (first) encodes permissions for 31-word segment (maximally) upstream.

One segment (last) encodes permissions for 32-word segment (maximally) downstream.

Total address range: 79 words

Advantage: much more compact

Advantage: overlap in segments may alleviate proximal loads from the table

2-bit entry type. Either pointer to next level, pointer to permission vector, or mini-SST entry.

Disadvantage: overlapping address ranges complicates table updates.
The PLB caches Permissions Table entries, analogous to the TLB.

Low order “don’t care” bits in the PLB tag increase the number of addresses a PLB entry matches, thus decreasing the PLB miss-rate.

Changes in permissions requires a PLB flush. As above, “don’t care” bits in the search key allow all PLB entries within the modified region to be invalidated during a single cycle.
Boothing Performance via 2-Level Permissions Caching

Each address register in the machine has an associated sidecar register.

On a PLB miss, the entry returned by the Permissions Table is also loaded into the appropriate sidecar register.

The base and bound of the user segment represented by the table entry are expanded to facilitate boundary checks.

Idea: the memory address referenced by a particular address register on the CPU will frequently load/store from/to that address or one within the same user segment, so hardwire the permissions.

Reduces traffic to the PLB.
Evaluating Performance Overhead

Evaluated both C and Java programs. (why?) that were a mix of both memory-reference and memory-allocation intensive.

One confounding parameter: the degree of granularity. Evaluated the extrema, (a) coarse-grained as provided by today’s VM, and (b) super-fine-grained where every object is its own user segment.

All benchmark programs run on a MIPS simulator modified to trace memory references.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Refs $\cdot 10^6$</th>
<th>Segments</th>
<th>Refs/Update</th>
<th>Cs</th>
</tr>
</thead>
<tbody>
<tr>
<td>crafty_test</td>
<td>3.088</td>
<td>96</td>
<td>64,327,162</td>
<td>6</td>
</tr>
<tr>
<td>gcc_tr</td>
<td>1.684</td>
<td>20,796</td>
<td>161,944</td>
<td>26</td>
</tr>
<tr>
<td>twolf_train</td>
<td>11,537</td>
<td>938,844</td>
<td>24,576</td>
<td>8</td>
</tr>
<tr>
<td>vpr_test</td>
<td>506</td>
<td>6,274</td>
<td>161,191</td>
<td>6</td>
</tr>
<tr>
<td>vortex_tr</td>
<td>1,291</td>
<td>211,630</td>
<td>12,200</td>
<td>16</td>
</tr>
<tr>
<td>j-compress</td>
<td>561</td>
<td>6,430</td>
<td>174,554</td>
<td>14</td>
</tr>
<tr>
<td>j-db</td>
<td>109</td>
<td>249,104</td>
<td>876</td>
<td>12</td>
</tr>
<tr>
<td>j-jack</td>
<td>402</td>
<td>1,622,330</td>
<td>496</td>
<td>34</td>
</tr>
<tr>
<td>j-jess</td>
<td>245</td>
<td>215,460</td>
<td>2,275</td>
<td>10</td>
</tr>
<tr>
<td>j-raytrace</td>
<td>1,596</td>
<td>1,243,032</td>
<td>2,567</td>
<td>20</td>
</tr>
<tr>
<td>m-jpeg_dec</td>
<td>1</td>
<td>58</td>
<td>45,785</td>
<td>6</td>
</tr>
<tr>
<td>m-mpeg2_dec</td>
<td>30</td>
<td>46</td>
<td>1,307,794</td>
<td>6</td>
</tr>
<tr>
<td>o-em3d</td>
<td>608</td>
<td>131,598</td>
<td>9,240</td>
<td>22</td>
</tr>
<tr>
<td>o-health</td>
<td>142</td>
<td>846,514</td>
<td>336</td>
<td>14</td>
</tr>
</tbody>
</table>

**Refs**: total no. of loads and stores $\times 10^6$

**Segs**: no. of segments written to PT

**R/U**: avg. references per PT update

**Cs**: no. of coarse-grained segments
Metrics

Space overhead = space occupied by protection tables ÷ by space being used by application (data + instructions) at end of run.

Runtime overhead = number of permissions table references (rw) ÷ number of memory references made by the application.

Caveat: not measuring peak overhead.

Space being used by application determined by querying every word in memory and seeing if it has valid permissions.

Caveat: this overhead may or may not manifest itself as performance loss, depending on cpu implementation.

Caveat: space between malloced regions not included in this quantity.
Coarse-Grained Protection Results

MLPT with mini-SST entries and 60-entry PLB versus conventional page table plus TLB.

Expectation: slight space overhead from MLPT leaf tables.

Expectation: slight speed improvement from additional hardware.

Claim: overhead for MMP word-level protection is very low when not used.

Expectations generally hold.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MLPT mSST 60 PLB</th>
<th>PAGE+TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X-ref</td>
<td>Space</td>
</tr>
<tr>
<td>crafty_test</td>
<td>0.56%</td>
<td>0.41%</td>
</tr>
<tr>
<td>gcc_tr</td>
<td>0.01%</td>
<td>0.08%</td>
</tr>
<tr>
<td>twolf_train</td>
<td>0.00%</td>
<td>0.31%</td>
</tr>
<tr>
<td>vpr_test</td>
<td>0.00%</td>
<td>0.62%</td>
</tr>
<tr>
<td>vortex_tr</td>
<td>0.02%</td>
<td>0.10%</td>
</tr>
<tr>
<td>j-compress</td>
<td>0.00%</td>
<td>0.11%</td>
</tr>
<tr>
<td>j-db</td>
<td>0.32%</td>
<td>0.17%</td>
</tr>
<tr>
<td>j-jack</td>
<td>0.00%</td>
<td>0.04%</td>
</tr>
<tr>
<td>j-jess</td>
<td>0.06%</td>
<td>0.18%</td>
</tr>
<tr>
<td>j-raytrace</td>
<td>0.00%</td>
<td>0.07%</td>
</tr>
<tr>
<td>m-jpeg_dec</td>
<td>0.27%</td>
<td>0.61%</td>
</tr>
<tr>
<td>m-mpeg2_dec</td>
<td>0.01%</td>
<td>0.61%</td>
</tr>
<tr>
<td>o-em3d</td>
<td>0.00%</td>
<td>0.07%</td>
</tr>
<tr>
<td>o-health</td>
<td>0.02%</td>
<td>0.12%</td>
</tr>
</tbody>
</table>
Removed permissions on malloc header and only allowed program access to the allocated block.

**Claim 1.** MLPT outperforms SST as segment number increases. Why?

**Claim 2.** MLPT space overhead is always < 9%.

**Claim 3.** The mSST table entry outperforms protection vectors.
Sidecar miss rate about 10-20%. PLB miss rate just 0.5%.

Impact of permissions table accesses on L1 L2 cache efficiency is slight, with less than an additional 0.25% being added to the miss rate in the worst case.
Conclusions

1. Fine-grained segment-based memory protection that is compatible with current linearly addressed ISAs is feasible.

2. The space and runtime overhead of providing this protection is small and scales with the degree of granularity.

3. The MMP facilities can be used to implement efficient applications.
Architectural Support for Single Address Space Operating Systems

Eric J. Koldinger, Jeffrey S. Chase and Susan J. Eggers
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
64-bit virtual address spaces are coming.

That’s more address space than a program could ever want or need.

This alleviates the existing evolutionary pressure on OSes to treat virtual addresses as a scarce resource that must be multiply allocated.

All programs can now live in one big happy address space. These are single address space (SAS) operating systems.

**Pro:** addresses are unique and context independent.

**Con:** no more private address space means no intrinsic protection.

This paper focuses on how to represent protection information in the cache structures in SAS systems.
The Promises of SAS OSes

**Support for Sharing**

- VAs are globally unique so can be passed between domains without translation.
- Alleviates the need for costly RPCs when communicating across protection domains.

**Virtually Indexed Caches**

- Virtually indexed caches are faster than physically indexed caches because no addy translation required.
- However, multiple address space OSes must use physical indexing because:
  - 2+ VAs from 2+ PDs may reference the same physical address (*synonyms*), causing coherency problems.
  - 1 VA from 2+ PDs may reference 2+ physical address (*homonyms*).

Both these may be circumvented, but at the cost of performance. In SAS systems, synonyms and homonyms don’t exist. Virtual to physical mapping is (can be) 1-to-1.
Motivation

We would like to take advantage of the benefits of SAS Oses.

To do so we need to restore the protection that we lost when we had a separate address space for every protection domain.

This paper seeks to evaluate two model of hardware support for protection in SAS systems.
What’s wrong with conventional architectures and SAS?

1. Protection domains in a SAS system would typically reference small and widely scattered pieces of the address space. Linear page tables cannot represent such sparse mappings compactly.

2. Translation mappings for shared pages must be duplicated in the page tables of for each domain. This is wasteful and invites coherency issues.
Two models for supporting protection in SAS systems

<table>
<thead>
<tr>
<th>Domain-Page Model</th>
<th>Page-Group Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies permissions explicitly for each domain-page pair.</td>
<td>Defines logical grouping of pages called <em>page-groups</em>.</td>
</tr>
<tr>
<td>Can be implemented by moving PD tags from the TLB to a <em>protection lookaside buffer (PLB)</em>.</td>
<td>A PD defined by the set of page-groups it can access.</td>
</tr>
<tr>
<td></td>
<td>Each page within a group has access rights that are used by all domains with access to the group.</td>
</tr>
</tbody>
</table>
Each PLB entry contains the protection information granted to one PD for one specific virtual page.

On each memory reference the PLB is accessed by the VPN and PD-ID, provided by processor ctrl register.

Note VA used for both cache and PLB, so lookups can occur in parallel.

Note that separation of translation and protection in this manner allows the PLB to be used in conjunction with a virtually indexed and tagged cache.
The PLB

Note this is different than what we’ve seen before. Address translation is outside the critical path of the cpu.

Here the TLB can be moved off-chip, allowing for potentially a much larger TLB.

Note the TLB only requires one entry for each virtual-to-physical mapping. A purge is required only on the change of a virtual-to-physical translation and not during a protection domain switch.
The Page-Group Model

This TLB takes a VPN and returns (a) a physical address, (b) rights, and (c) an access identifier (AID) that contains a page-group number.

The processor must determine whether the current PD has access to the page-group identified by the AID.

Four page-group registers (PID) store the set of page-groups accessible to the current PD.

If AID == 0 (global) or AID == PID_{1-4} then access is granted, with rights given by (a) the TLB, (b) the current CPU privilege level, and (c) a write bit.
Note 1 if access is not granted then an access violation is signaled and the kernel is invoked.

Note 2 the four page-group registers obviously limit the number of groups a PD can access. For eval, the authors assume an LRU cache of page-groups.

Note 3 translation and protection are combined in this TLB, thus the TLB must be on-chip. But a virtually indexed TLB and on-chip PLB could have been used as well, thus making page-grouping a bit of an orthologous issue.
<table>
<thead>
<tr>
<th>Application Type</th>
<th>Action and Description</th>
<th>Frequency per Domain</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>Attach Segment</td>
<td>Once per segment</td>
<td>Allow access rights to be faulted into the PLB, one page at a time</td>
</tr>
<tr>
<td></td>
<td>Detach Segment</td>
<td>Once per segment</td>
<td>Purge the PLB or inspect each entry and eliminate those for the segment-domain pair affected</td>
</tr>
<tr>
<td>Concurrent Garbage</td>
<td>Flip Spaces Change to-space to from-space. Create new segment for new to-space. Make both spaces read-write for the collector only</td>
<td>Once per garbage collection</td>
<td>Remove the appropriate page-group identifier from the page-group cache</td>
</tr>
<tr>
<td>Collection [2]</td>
<td>Access un-scanned to space: Trap the access, garbage collect the page and move it to &quot;scanned&quot; to-space, making it read-write for the application</td>
<td>Once per page touched</td>
<td>Inspect each entry in the PLB, marking those for from-space as no access for the application</td>
</tr>
<tr>
<td>Distributed VM [7, 30]</td>
<td>Get Readable: Trap the access, get a readable copy of the page and make it read-only</td>
<td>Once per access of the remote page</td>
<td>Garbage collect the page. Place in to-space page-group for the application domain</td>
</tr>
<tr>
<td></td>
<td>Get Writable: Trap the access, get an exclusive copy of the page and make it read-write</td>
<td>Once per access of the remote page</td>
<td>Check to see if the copy in memory is valid, and retrieve it from the remote host if it's not. Set read-only rights in the PLB.</td>
</tr>
<tr>
<td></td>
<td>Invalidate: A remote machine invalidates the page. Make it inaccessible on this node.</td>
<td>Once per remote access to the local page</td>
<td>Check to see if the copy in memory is valid; retrieve it from the remote host if it's not. Invalidate any other remote copies. Set read-write access in the PLB.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Set access rights to none in the PLB.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Set access rights to none in the TLB.</td>
</tr>
</tbody>
</table>
### Evaluation B

<table>
<thead>
<tr>
<th>Transactional VM [9, 16]</th>
<th><strong>Lock (read):</strong> Allow shared, read-only access</th>
<th>Once per page touched, per transaction</th>
<th>Determine if the page can be locked; if so, mark it as locked and set the read bit in the PLB entry for transaction’s domain</th>
<th>Determine if the page can be locked; if so, mark it as locked. Determine the correct page-group for the pages locked by the current domain, and move this page to that page group. Set the access rights to allow reading.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lock (write):</strong> Allow private, read-write access</td>
<td>Once per page touched, per transaction</td>
<td>As above, except set both the read and write bits in the PLB entry for the transaction’s domain</td>
<td>As above, except set the access rights to allow both reading and writing.</td>
<td></td>
</tr>
<tr>
<td>Commit: Unlock all locked pages and return them to the inaccessible state</td>
<td>Once per page touched, per transaction</td>
<td>Remove all the locks held by this transaction. For each locked page, look up the page in the PLB, and change the access rights to inaccessible, or change the PD-ID to represent the new transaction</td>
<td>Remove all the locks held by this transaction. For each page that was locked, look up in the TLB and move to inaccessible page-group; or remove lock groups from the page-group cache and allocate new groups for the next transaction’s locks.</td>
<td></td>
</tr>
<tr>
<td>Concurrent Checkpoint [31]</td>
<td><strong>Restrict Access.</strong> Remove clients’ access rights to all pages in the segment</td>
<td>Once per checkpoint taken</td>
<td>Inspect each entry in the PLB and mark the pages as read-only for the application</td>
<td>Mark the page-group for this segment as read-only to the application. Allocate a different group as read-write for this segment for both the application and server.</td>
</tr>
<tr>
<td><strong>Checkpoint Page.</strong> Trap the access and write the page to disk. Make the page read-write for the application</td>
<td>Once per page, during the checkpoint operation</td>
<td>Write the page to disk. In the PLB mark it as read-write for the application</td>
<td>Write the page to disk. Move it to a new read-write group in the TLB.</td>
<td></td>
</tr>
<tr>
<td>Compression Paging [3]</td>
<td><strong>Page-out:</strong> Make the page inaccessible to the application, compress the data on the page, write it to disk and unmap the page</td>
<td>Every time a page is deallocated</td>
<td>Mark the page inaccessible to the client in the PLB. Compress the data on the page and write it to disk. Remove the page entry from the TLB and allow the page to be reallocated</td>
<td>Move the page to the page-group private to the server in the TLB. Compress the data on the page and write it to disk. Remove the page entry from the TLB and allow the page to be reallocated.</td>
</tr>
<tr>
<td><strong>Page-in:</strong> Allocate the physical page, read data from the disk and decompress. Make the page accessible to the client</td>
<td>Every time a page is brought in from secondary storage.</td>
<td>Allocate the physical page, map it in the TLB, and mark it accessible to the server in the PLB. Read the page and decompress the data. Make the page accessible to the client in the PLB</td>
<td>Allocate the physical page, map it in the TLB, and put it in the server’s private page-group with read-write access. Read the page and decompress the data. Move to page-group representing this segment for the client.</td>
<td></td>
</tr>
</tbody>
</table>