Stupid Pet Tricks with Virtual Memory

Robert Grimm
New York University
The Three Questions

- What is the problem?
- What is new or different?
- What are the contributions and limitations?
How to control VM: Segmentation fault handler

- Override "segmentation fault: core dumped" message
  - A rather coarse hook, which doesn’t even have to be efficient

Yet many applications can benefit from VM support

- General idea: detect reads/writes
  - No changes to compiler to include checks
  - No runtime overhead for explicit checks in code
Virtual Memory Primitives

- **Trap**: Handle page faults in user mode
- **Prot1**: Decrease accessibility of one page
- **ProtN**: Decrease accessibility of N pages
  - More efficient than calling Prot1 N times
- **Unprot**: Increase accessibility of page
- **Dirty**: Return list of pages written to since last call
  - Can be emulated with ProtN, Trap, and Unprot
- **Map2**: Map physical page to different virtual pages
  - At different access levels, in the same address space
Application of Primitives

<table>
<thead>
<tr>
<th>Methods</th>
<th>TRAP</th>
<th>PROT1</th>
<th>PROT2</th>
<th>UNPROT</th>
<th>MAP2</th>
<th>DIRTY</th>
<th>PAGESIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrent GC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SVM</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Concurrent checkpoint</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generational GC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Persistent store</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Extending addressability</td>
<td>✓</td>
<td>*</td>
<td>*</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Data-compression paging</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Heap overflow</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

* All applications besides heap overflow detection use a combination of several primitives

* Many applications can benefit from control over page size
Concurrent GC

- Concurrent threads
  - Mutator(s) do actual work (including allocations)
  - One collector reclaims unused memory
- Two memory halves: from-space, to-space
  - Collector copies used data from from-space to to-space
  - Pointers in from-space are *forwarded* to to-space
- Algorithm invariants
  - Mutator sees only to-space pointers in registers
  - Objects in new area contain only to-space pointers
  - Objects in scanned area contain only to-space pointers
  - Objects in unscanned area may contain pointers to both
Concurrent GC (cont.)

- First invariant requires checks on every access
  - Protect from-space: no access for unscanned data
  - Trap causes collector to fix invariant
    - Triggered by mutator accessing data
- Algorithm requires Trap, ProtN, Unprot, Map2
  - Trap to detect fetches from unscanned data
  - ProtN to flip protection on spaces
  - Unprot to release scanned pages
  - Map2 to access unscanned data through collector
- Small page size to reduce collector latency
Shared Virtual Memory

- Same virtual address space for several CPUs
- Local memory as cache
- Read-only pages may be shared
- But, writable pages may only be present on one node
  - Use Trap, Prot1, and Unprot
- Small page size — why?
Concurrent Checkpointing

- Goal: save all memory contents to disk
- Problem: stop and save takes too long
- Solution
  - Entire address space marked as read-only
  - Copying thread scans address space
    - Restores write access after copying page
    - Gives precedence to write-faulting pages
  - Requires Trap, Prot1, ProtN, Unprot, Dirty
    - Prot1 and Dirty used for incremental checkpoints
  - Suggests medium page size — why?
Generational GC

Two motivating observations

- Younger records die much sooner than older records
- Younger records point to older records, but older records don’t usually point to younger records

GC strategy: divide heap into several *generations*

- Perform GC at the granularity of generations
  - More frequently for younger generations
- Problem: detecting writes to older generations
  - 5-10% overhead when using instructions
- Solution: detect modified pages in older generations
  - Dirty or (Trap, ProtN, Unprot) with a small page size — why?
Persistent Stores

Basic idea: persistent object heap

- Modifications can be committed or aborted
- Object accesses are (almost) as fast as regular memory
  - Compare with traditional databases

Implementation strategy

- Database is a memory mapped file
  - Pointer traversals run at memory speeds
- Uncommitted writes are temporary up to commit

Requirements

- Trap, Unprot, file mapping with copy-on-write
  - Copy-on-write can be simulated with ProtN, Unprot, Map2
More Applications

- Extending addressability
  - Convert between different pointer resolutions
    - Pointed-to pages protected by virtual memory
    - Page table entry contains address of disk page
- Data-compression paging
  - Store compressed version in memory instead of disk paging
    - Integration with GC avoids overhead
- Stack & heap overflow detection
  - Mark page above stack/heap as no access
    - Least interesting of examples; well-known technique
Evaluation & Discussion
Virtual Memory Performance

* Two categories
  * ProtN, Trap, Unprot
  * Prot1, Trap, Unprot

* Wide variation
  * In performance, even on same hw
  * In VM API correctness
    * shmat on Ultrix
    * mprotect
System Design Issues

- TLB consistency on multiprocessors
  - Need to shoot down TLB entries when changing protection
    - Cannot allow temporary inconsistency
  - Need to batch shoot-downs for efficiency
    - Can also benefit regular disk paging

- Optimal page size
  - Hard to reconcile VM applications with paging
    - One possible solutions: pages vs. multi-page blocks
More Issues

- Access to protected pages
  - Service routine needs to access page while client cannot
    - Here’s a bunch of possible solutions... but...
  - For physically addressed caches: Map2 is a good solution
  - For virtually addressed caches: Potential of inconsistency
    - Though not a problem for concurrent GC
      - Mutator has no access to page
        - Therefore no cache lines present for mutator
      - After scanning page, collector flushes cache
The Trouble with Pipelining

- **Problem**
  - There may be several outstanding page faults
  - Instructions *after* faulting one may have already stored their results in registers
  - Instructions can be *resumed* but *not* restarted

- **Observation:** most algorithms sufficiently asynchronous
  - Comparable to traditional disk-pager
    - Get fault, provide page, make page accessible, resume
  - Exception: heap overflow detection
    - Combine limit check for several allocations with unrolled loop
What Do You Think?