Cool Pet Tricks with...

...Virtual Memory

Robert Grimm
New York University
Altogether Now: The Three Questions

- What is the problem?
- What is notable?
- What are the contributions and limitations?
Motivation

- Application control: Segmentation fault handler
  - Override “segmentation fault: core dumped” message
  - Neither expressivity of VM interface nor performance matters

- But, there are many applications that benefit from virtual memory 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-fault traps in user mode
- Prot1: Decrease accessibility of a page
- ProtN: Decrease accessibility of N pages
  - More efficient than calling Prot1 N times
- Unprot: Increase accessibility of a page
- Dirty: Return list of written pages since last call
  - Can be emulated with ProtN, Trap, and Unprot
- Map2: Map same physical page at two different virtual addresses, at different access levels, in the same address space
Virtual Memory Primitives and Their Applications

- All applications besides heap overflow detection use a combination of several primitives
- Many applications can benefit from control over pagesize

<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>
Concurrent Garbage Collection

- 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 \textit{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 Garbage Collection (cont.)

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

- Same virtual address space for several CPUs
  - Local memory serves 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 pagesize (why?)
Generational Garbage Collection

- 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 with instructions is slow (5-10% of execution time)
  - Solution: Detect modified pages in older generations
    - Dirty or (Trap, ProtN, Unprot)
    - Small pagesize (why?)
Persistent Stores

- **Basic idea: Persistent object heap**
  - Modifications can be committed or aborted
  - Advantage over traditional databases: object accesses are (almost) as fast as regular memory accesses

- **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
    - Can be simulated through ProtN, Unprot, Map2
More Applications

- **Extending Addressability**
  - Basic idea: Convert between different pointer resolutions
    - Pointed-to pages protected by virtual memory
    - Page table entry contains address of disk page

- **Data-Compression Paging**
  - Basic idea: Store compressed version in memory instead of disk paging
    - Integration with garbage collector avoids overhead

- **Stack & heap overflow detection**
  - Basic idea: Mark page above stack or heap as no access
    - Least interesting of examples, well-known optimizations
Evaluation & Discussion
Virtual Memory Performance

- Two categories
  - ProtN, Trap, Unprot
  - Prot1, Trap, Unprot
- Wide variation
  - In performance, even on same hardware
  - In VM API correctness
    - shmat on Ultrix
    - mprotect

![Figure 2: Instructions per PROT + TRAP + UNPROT.](chart)
System Design Issues

- **TLB consistency on multiprocessors**
  - Need to shoot down TLB entries when making page less accessible
    - Cannot allow temporary inconsistency
  - Batch shoot downs for efficiency (hence ProtN)
    - Can also benefit regular disk paging

- **Optimal page size**
  - Hard to reconcile VM applications with paging
    - One possible solution: pages vs. multi-page blocks
Access to protected pages

- Service routine needs to access page while client can not
  - We can come up with lots of solutions, but here are the good ones…
- For physically addressed caches: Map2 is a good solution
- For virtually addressed caches: Potential of cache inconsistency
  - For concurrent garbage collection
    - Mutator has no access to page (by definition of algorithm)
    - Therefore, no cache lines present for mutator
    - After scanning page, collector needs to flush 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 are sufficiently asynchronous
  - Comparable to traditional disk-pager
    - Get fault, provide page, make page accessible, resume
  - **Exception:** Heap overflow detection
    - Better strategy: Combine limit check for several allocations with unrolled loop
What Do You Think?