Virtualization

Part 2 – VMware
VMware: binary translation

References and Sources

Binary Translation

Characteristics
- Binary – input is machine-level code
- Dynamic – occurs at runtime
- On demand – code translated when needed for execution
- System level – makes no assumption about guest code
- Subsetting – translates from full instruction set to safe subset
- Adaptive – adjust code based on guest behavior to achieve efficiency
Binary Translation

TU: translation unit (usually a basic block)
CCF: compiled code fragment
Continuation: continuation

Hash Table

Translation Cache

PC → [x] → Binary Translator → [y] → execute

Few cache hits
Working set captured

Running time

% translation
Eliminating faults/traps

- Expensive traps/faults can be avoided
- Example: Pentium privileged instruction (rdtsc)
  - Trap-and-emulate: 2030 cycles
  - Callout-and-emulate: 1254 cycles
  - In-TC emulation: 216 cycles

- Process
  - Privileged instructions – eliminated by simple binary translation (BT)
  - Non-privileged instructions – eliminated by adaptive BT
    - (a) detect a CCF containing an instruction that trap frequently
    - (b) generate a new translation of the CCF to avoid the trap (perhaps inserting a call-out to an interpreter), and patch the original translation to execute the new translation
Memory resource management

- VMM (meta-level) memory management
  - Must identify both VM and pages within VM to replace
  - VMM replacement decisions may have unintended interactions with GuestOS page replacement policy
  - Worst-case scenario: double paging

- Strategies
  - "ballooning" –
    - add memory demands on GuestOS so that the GuestOS decides which pages to replace
    - Also used in Xen
  - Eliminating duplicate pages – even identical pages across different GuestOSs.
    - VMM has sufficient perspective
    - Clear savings when running numerous copies of same GuestOS
  - Allocation algorithm
    - Balances memory utilization vs. performance isolation guarantees
    - “taxes” idle memory
Ballooning

- “balloon” – module inserted into GuestOS as pseudo-device driver or kernel service
- Has no interface to GuestOS or applications
- Has a private channel for communication to VMM
- Polls VMM for current “balloon” size
- Balloon holds number of “pinned” page frames equal to its current size
- Inflating the balloon
  - Balloon requests additional “pinned” pages from GuestOS
  - Inflating the balloon causes GuestOS to select pages to be replaced using GuestOS page replacement policy
  - Balloon informs VMM of which physical page frames it has been allocated
  - VMM frees the machine page frames corresponding to the physical page frames allocated to the balloon (thus freeing machine memory to allocate to other GuestOSs)
- Deflating the balloon
  - VMM reclaims machine page frames
  - VMM communicates to balloon
  - Balloon unpins/frees physical page frames corresponding to new machine page frames
  - GuestOS uses its page replacement policy to page in needed pages
Content-based page sharing

- A hash table contains entries for shared pages already marked “copy-on-write”
- A key for a candidate page is generated from a hash value of the page’s contents
- A full comparison is made between the candidate page and a page with a matching key value
- Pages that match are shared – the page table entries for their VMMs point to the same machine page
- If no match is found, a “hint” frame is added to the hash table for possible future matches
- Writing to a shared page causes a page fault which causes a separate copy to be created for the writing GuestOS
Page sharing performance

- Identical Linux systems running same benchmark
- “best case” scenario
- Large fraction (67%) of memory sharable
- Considerable amount and percent of memory reclaimed
- Aggregate system throughput essentially unaffected
Measuring Cross-VM memory usage

- Each GuestOS is given a number of shares, \( S \), against the total available machine memory.
- The shares-per-page represents the “price” that a GuestOS is willing to pay for a page of memory.
- The price is determined as follows:

\[
\rho = \frac{S}{P \cdot (f + k \cdot (1 - f))}
\]

- The idle page cost is \( k = 1/(1-\tau) \) where \( 0 \leq \tau < 1 \) is the “tax rate” that defaults to 0.75.
- The fractional usage, \( f \), is determined by sampling (what fraction of 100 randomly selected pages are accesses in each 30 second period) and smoothing (using three different weights).
Memory tax experiment

- Initially, VM1 and VM2 converge to same memory allocation with $\tau=0$ (no idle memory tax) despite greater need for memory by VM2.
- When idle memory tax applied at default level (75%), VM1 relinquishes memory to VM2 which improves performance of VM2 by over 30%.