CSCI-GA.3033-015

Virtual Machines: Concepts & Applications

Lecture 5: Case Studies of Process VMs

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Disclaimer: Many slides of this lecture are based on the slides of authors of the textbook from Elsevier. All copyrights reserved.
SUN Shade
Case Study: SUN Shade

- A fast instruction set simulator
- Uses binary translation for speed
  - uncommon in simulators
  - typically interpretation is used
- Shades also generates traces
  - Needed for the simulation but not the emulation
Shade Code Structures

- VMEM
- Text
- Data
- Stack

- TLB: Translation Lookaside Buffer (Hash Table)
- TC: Translation Cache
- vpc: virtual (source) PC; held in a register

Array maintained by shade
Shade Translations

- Translation Units
  - Basic block or superblock
  - Some source code fragments may be in more than one translation block
- Stops at special instructions
  - Synchronization, indirect branches, software traps
  - Limit on translation block size
    - To simplify storage allocation
- Translation Cache Size 4MB
Translation Cache

• Translated blocks chained to decrease TLB look-up
• Translations fills the TC linearly as they are produced.
• Flushed when full
Shade TLB (Hash Table)

- Maps source PC to cached translation (unfortunate, confusing name)
- An array of lists (e.g. n-way set assoc)
  - Each list is n <source,target> pairs
- TLB Size 256KB (32K entries)
- Put the last-met translation (i.e. most recently used) first in the list
- If list full → replace the right-most

![Diagram showing the Shade TLB with hash table and list structure]

source PC → hash → target pc

<table>
<thead>
<tr>
<th>source</th>
<th>target</th>
<th>source</th>
<th>target</th>
<th>...</th>
<th>source</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>target</td>
<td>source</td>
<td>target</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>source</td>
<td>target</td>
<td>source</td>
<td>target</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>source</td>
<td>target</td>
<td>source</td>
<td>target</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>source</td>
<td>target</td>
<td>source</td>
<td>target</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>source</td>
<td>target</td>
<td>source</td>
<td>target</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Shade TLB (Hash Table)

• When an entry is replaced, and “orphan” translation may be formed in the TC.
  – May force redundant re-translation
  – Flush when full deals with that

• Align TLB lines with cache boundaries
Shade Main Simulation Loop

- Hashes vpc (source PC) to TLB array index
- Fast partial search of TLB entry for vpc value
- If hit, use target translation
- If miss, call function to do full search of TLB entries
  - If hit, move hit entry to front of hash list
  - If miss, generate translation, place at front of list
- Finding Successor block
  - Code block places next source PC in vpc then jumps to VMM
Shade Chaining Setup

- Predecessor translated first:
  - Block finishes with BAL to VMM
  - Translator overwrites BAL with branch to successor
- Indirect Jump Chaining:
  - Not used; always return to VMM
FX!32

IA-32 → 64-bit Alpha platform running Windows
Case Study: FX!32

- **Runtime software**
  - Follows typical model
  - But, translations/optimizations are done between executions
    - First execution of binary: interpret and profile
    - Translate and optimize “off line”
    - Later execution(s): use translated version, continue profiling
      - Fast interpretation is crucial here!

- **Persistence**
  - Translations and profile data are saved on disk between runs
FX!32 Overview

Diagram:
- Transparency Agent
- Binary Memory Image
- Interpreter/Emulator
- Translated Images
- Profile Data
- Translator
- Database (Registry)
- Server
Transparency Agent

• **Main job:** provide seamless (transparent) integration of host and user processes

• This is done through: **encapsulation**
Encapsulation

- Guest code is “encapsulated”
  - At creation by loader
- Creation
  - Host can create guest
  - Guest can create host
- DLLs
  - Guest can use guest or host
  - Host uses only host

How does the loader differentiate guest process from host process?
Loaders

• Requires two loaders
  – One for host processes
  – One for guest processes

• How can we invoke the correct one? → 3 approaches
  – Modify kernel loader
    • Identifies type of binary, calls correct loader
    • Requires modification of kernel loader
  – Add code to guest binary when installed
    • Invokes guest loader
    • Requires local installation of guest binary
  – Modify host process create_process API
    • Invokes guest loader for guest binaries
    • Modifies create_process in host binaries
    • Used in FX!32
Transparency Agent

- “Hooked” onto Windows `CreateProcess` API routine
- Examines every image about to be executed
  - If x86 image, invokes FX!32 runtime
- A process containing the transparency agent is enabled
- Each attempt to launch an Alpha process results in that process being enabled
- Windows shell, service control manager, remote procedure call server
  - Then, essentially spreads like a virus
Runtime

• Sets up runtime environment
• Calls emulator (interpreter)
• Registers image with FX!32 database
  – Database associates identifier with translations and profiles
  – Identifier is a hash of the image itself
  – Given an image, runtime hashes it, and looks it up
  • If found, it can read in the translation and profile data
Interpreter

• Used for initial execution of image
• Also later executions for undiscovered code

• Register mapping:
  – X86 registers permanently mapped to Alpha registers
    • Many more alpha regs than x86 regs
    • X86 condition codes also held in alpha registers
Interpreter

• High performance
• Decode & dispatch
• 32-bit IA-32 mapped to 64-bit Alpha platform
• Written in assembly
• Optimized using software pipelining
• Fits in L1 instruction cache
Profile Generation

- Part of interpretation process
- Collects:
  - Call target addresses
  - \(<\text{Source address, Target address}>\) pairs for indirect control transfers
  - ...
- Hash table
  - Used for collecting profile data
  - When execution ends, runtime processes hash table to produce profile file
  - Hash table also used to detect already translated code
    - \(<\text{x86 address, translation address pairs}>\) are in table
  - Hash table loaded when translated image is loaded
Translator

- Invoked by the server “offline” as a background task
- Uses profile information to translate x86 images to alpha code
- Growth in amount of profile data causes re-translation
  - As the emulator finds previously unreached portions of x86 code
- Contains 8 main components:
  1. regionizer
  2. build
  3. Register mangler
  4. Condition code mangler
  5. Improve
  6. Code selector
  7. Code scheduler
  8. Code assembler
Translator: Regionizer

- Divides x86 image into *routines*
- Constructed by following control flow
  - Each call target is a routine entry point
  - Indirect jumps use profile targets from hash table
- A routine is a collection of *regions*
  - Contiguous instructions reachable from routine entry
  - A routine is the smallest collection of reachable regions
  - Many routines have a single region
Translator: Build

- Applied to each routine
- Re-parses x86 instructions
- Forms internal representation (IR)
  - Similar to IRs used by compilers
Translator: Register Mangler

- IR uses single assignment form for registers
- X86 allow partial register reads/writes
- Register mangler puts in insert/extract operations to deal with partial register accesses
Translator: Condition Code Mangler

- Condition code results are implicit in IR
- Mangler puts in instructions to explicitly generate CC values
- Keeps track of live CCs
  - Only generates code for those that are eventually used
Translator: Improve

• Basic block level optimizations
• Combine stack operations
  – Replace individual pointer decrements/increments for each individual stack operation
  – Combine all decrements/increments for a basic block
    • Perform single stack pointer subtract at basic block beginning
    • And single stack pointer add at end
• Eliminate stack loads and stores
Translator: Code Selector, Scheduler, Assembler

- Transforms IR to alpha code
- Each x86 instruction is mapped to one or more alpha insts
- Instructions are scheduled for Alpha pipeline
- Final translated image is built
ISA Issues: x86 CC handling

• X86 uses Condition Codes; Alpha does not
  – Use alpha registers to hold x86 CCs

• Condition Codes are *implicit* in x86
  – Frequently set
  – Infrequently used

• Interpreter uses lazy evaluation
  – Save enough state to allow CC evaluation if needed later
ISA Issues: Floating Point Instructions

- x86 uses 80 bit floating point for intermediate values
  - Alpha (and most other machines) use 64
  - Emulator uses only 64
    - Good performance
    - Very few applications rely on full 80 bits

- X86 uses stack; sometimes in strange ways
  - Interpreter optimized for strict stack behavior
ISA Translation Issues: Alignment

- X86 allows unaligned memory accesses
- Alpha accesses must be word aligned
  - Normal load/store with unaligned addresses cause trap to fixup code
  - Faster alternative: use multiple instruction sequences
    - But slower than access to known-aligned address
- Code selector uses profile information to find load/stores that have accessed unaligned data
  - Inserts proper multi-instruction sequence for these
  - Reduces number of traps to fixup code
Performance

• (comparing 200 MHz Pentium Pro and 500 MHz 21164)
• Goal: same as high-end x86
• Goal achieved for many integer benchmark
• Flt point – 30% slower than Pentium Pro
• Achieves 70% of native alpha performance
SUN Wabi

MS Windows Win16 API → Solaris
SUN Wabi: Windows ABI

- Runs *common* Windows applications on Unix systems
  - Either intel or non-intel platforms
  - Translates Windows calls to Unix calls

- **Goals/Objectives**
  - Speed over space (within reason)
  - Small footprint
  - x86 little endian regardless of host
  - 64-bit FP
**Wabi: Design Issues**

- **Uses Dynamic Translation vs. FX!32**
  - No persistence between executions
  - No extra disk space (3x in FX!32)
  - Limited time to generate code
- **Translate on first code reference**
  - Does not switch between interpretation and translation
- **Written in C (gcc)**
  - Can be ported to multiple platforms
Translator

• Always translate; “fast and dumb”, per interval
  – Interval: augmented basic block
  – May include simple loops
  – Multiple terminating conditional branches (superblocks)

• Register mapping
  – All x86 registers mapped to SPARC registers
  – x86 CCs mapped to SPARC CCs

• CC Optimization
  – Dead CCs detected
  – Lazy evaluation of CCs
Translator, contd.

- **Code Cache**
  - 1-64Mb, default = physical memory/4 +8MB
  - FIFO management

- **Chaining**
  - Most intervals are chained
  - Indirect transfers use hash table
  - Tables start small and grow based on loading factor
Translator, contd.

- X86 peephole optimization

  xor  ax,ax ⇒ clr ax  
  or   ax,ax ⇒ tst ax  
  jcc  lab1  
  jmp  lab2 ⇒ revjcc lab2  

Lab1:

- Inline small functions
- Reallocate x86 memory data to SPARC registers/memory
- Use adaptive optimization for high frequency intervals
Performance Problems

• Detecting and reporting x86 exceptions is expensive
  – Limits optimization and instruction scheduling
  – Requires memory to recover state info
  – fpu implementation requires checks in generated code

• Shortage of host registers for both x86 and emulator state
Dynamo

http://www.dynamorio.org/
Same-ISA Optimization

- **Objective**: Optimize binaries on-the-fly
  - Many binaries are un-optimized or are at a low optimization level
- **Initial emulation can be done very efficiently**
  - “Translation” at basic block level is identity translation
  - Initial sample-based profiling is attractive
    - Original code can be used, running at native speeds
Same-ISA Optimization

• Code patching can be used
  – Patch code cache regions into original code
  – Replace original code with branches into code cache (saves code some code duplication)

• Can bail-out if performance is lost
Code Patching Example
Dynamo

- Maps HP-PA ISA onto itself (newer version uses IA-32)
- Improved optimization is goal

```
interpret until taken branch

lookup branch target in cache

start-of-trace condition?

jump to top of fragment in cache

increment counter assoc. with branch target addr

counter value exceeds hot threshold?

interpret + codegen until taken branch

end-of-trace condition?

no yes hit

no yes

signal handler OS signal

emit into cache, link with other fragments & recycle the associated counter

create new fragment and optimize it

Fragment Cache

no

increment counter assoc. with branch target addr

signal handler OS signal
```

The diagram illustrates the process flow of Dynamo, including
- Native instruction stream
- Interpretation until taken branch
- Lookup branch target in cache
- Start-of-trace condition decision
  - Yes: Increment counter with branch target address
  - No: Counter value exceeds hot threshold?
    - Yes: Interpret + codegen until taken branch
    - No: End-of-trace condition?
      - Yes: Interpret + codegen until taken branch
      - No: Interpret until taken branch
- Fragment Cache
  - Jump to top of fragment in cache
  - Emit into cache, link with other fragments & recycle the associated counter
  - Create new fragment and optimize it
```
Basic Operation

- Staged optimization
  - Beginning with interpretation
  - Then optimized translations
- Translation unit: Fragment
  - Dynamic superblock; can contain calls/returns
  - Start points:
    - Backward taken branches
    - Fragment cache exit branches
  - End points:
    - Backward taken branches
    - Taken branch with target in fragment cache
    - Reaches maximum length
- Avg. overhead less than 1.5% of execution time
Fragment Cache Mapping

- Implementing replacement alg. is time consuming
  - Especially with a high level of internal linking
- Uses pre-emptive flushing technique
  - When new fragment creation rises sharply (working set change)
  - Flush entire cache (low overhead)
  - Good idea, if indeed the working set is changing
  - Also adapts to changes in branch bias
Performance

- Outperforms +O2; +O4, but not +O4 plus profiling
  - This may be due to code layout
  - Many app developers do not profile
Conclusions

• Most process VMs are built with both interpretation and translation
• The main difference is when to use each
• To reduce performance loss:
  – code cache
  – translated code optimization
  – optimization of interpretation