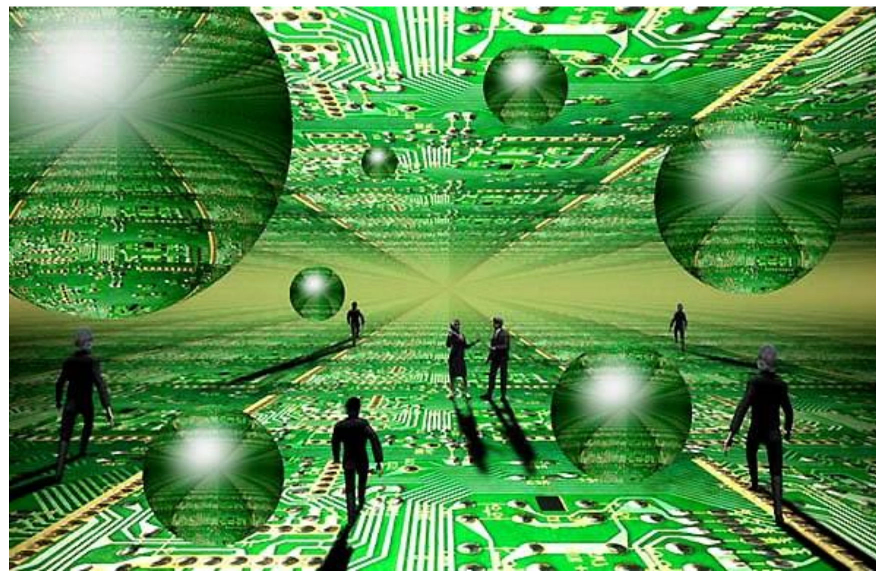
CSCI-GA.3033-016

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

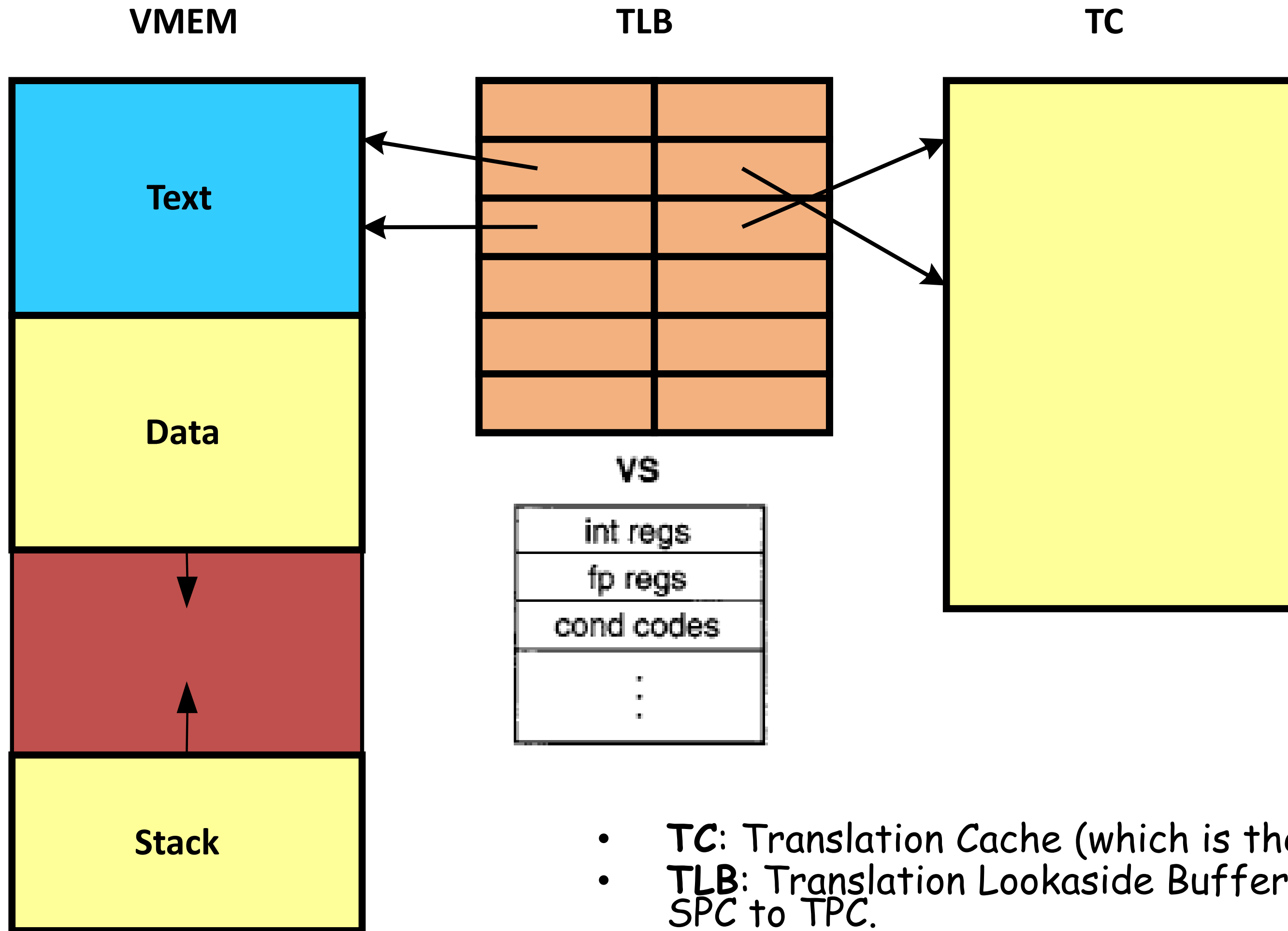
Host: SPARC

Guest: SPARC or MIPS

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



- **TC**: Translation Cache (which is the code cache we saw)
- **TLB**: Translation Lookaside Buffer (Hash Table) for mapping SPC to TPC.
- **vpc**: virtual (source) PC; held in a register
- **VS**: contains registers state

Shade Translations

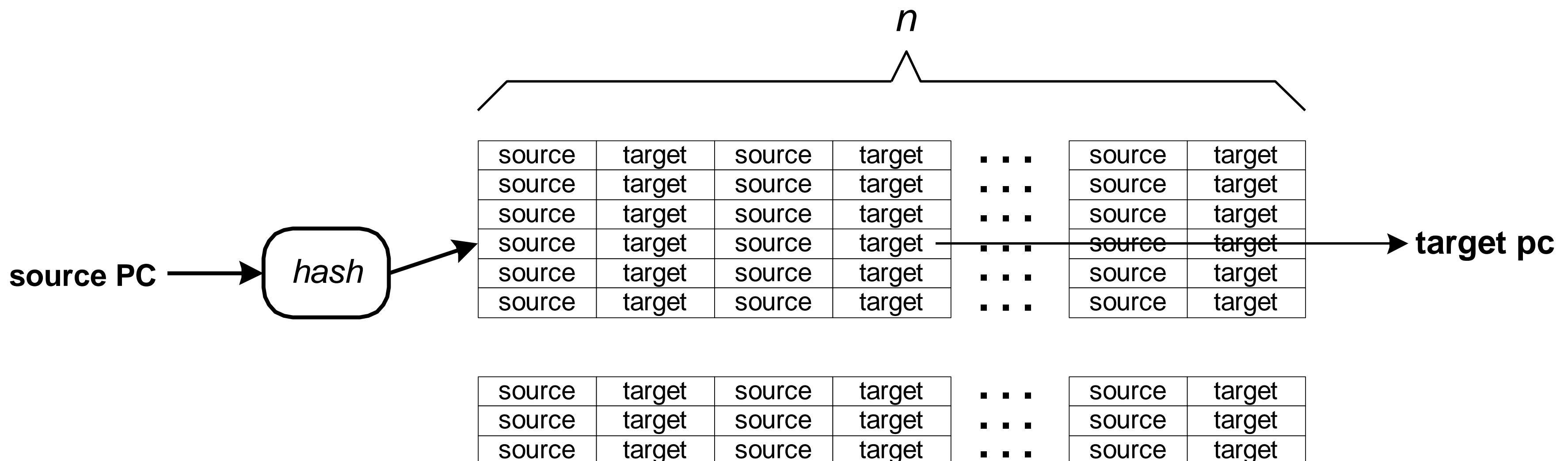
- Translation Units
  - Basic block or superblock
- Stops at special instructions
  - Synchronization, indirect branches, software traps
  - Limit on translation block size
    - To simplify storage allocation
- Translation Cache Size 4MB

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)
- A 2D array (n <source, target> pair per line)
  - n <source, target> pair per line
- 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



Shade TLB (Hash Table)

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

Shade Main Simulation Loop

- Hashes vpc (source PC) to TLB array index
- Shade checks the first translation of TLB row for vpc value
- If hit, use target translation
- If miss, call function to do full search of TLB entries in the row
  - If hit, move hit entry to front of hash list
  - If miss, generate translation, place at front of list

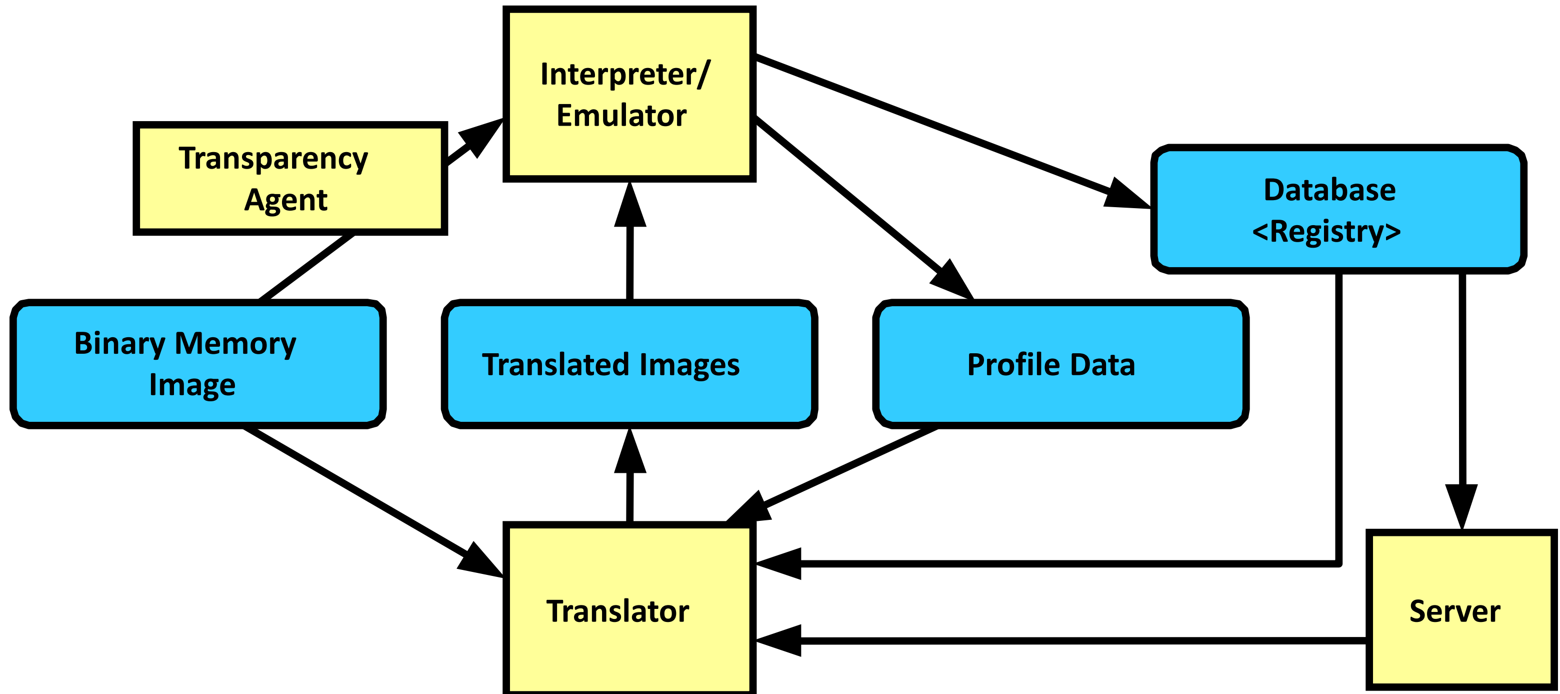
FX!32

IA-32 → 64-bit Alpha platform running Windows

Case Study: FX!32

- Runtime software
  - Translations/optimizations are done between executions
    - First execution of binary: interpret and profile
    - Translate and optimize "offline"
    - 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

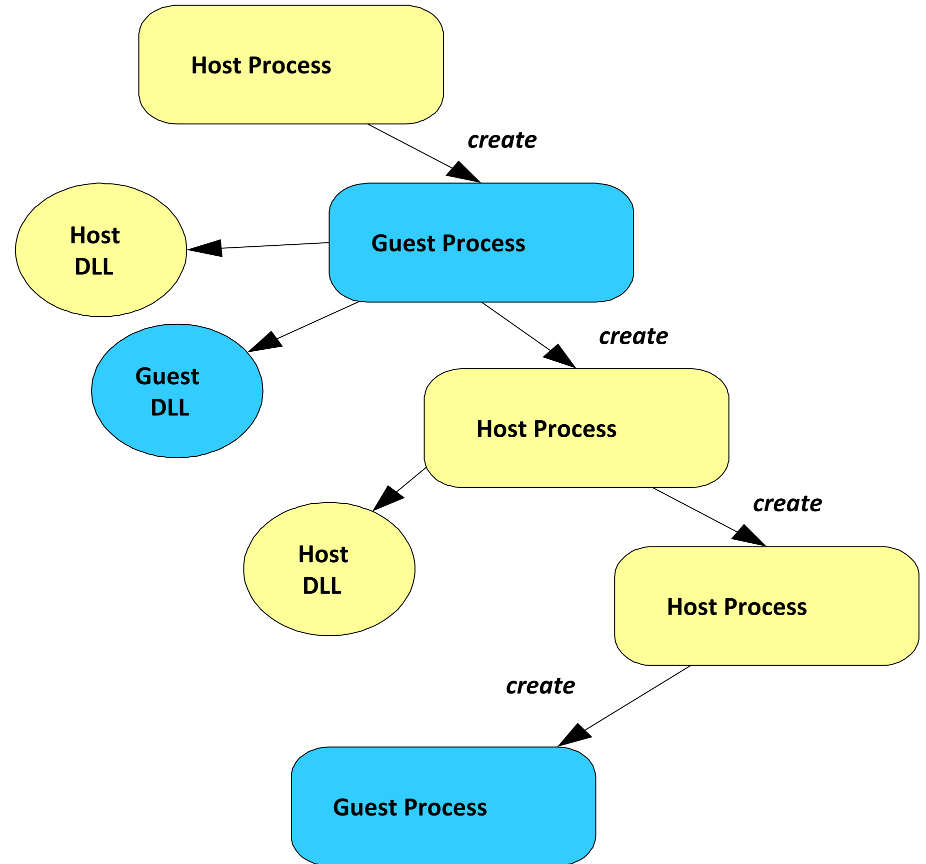


Transparency Agent

- 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
  - 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

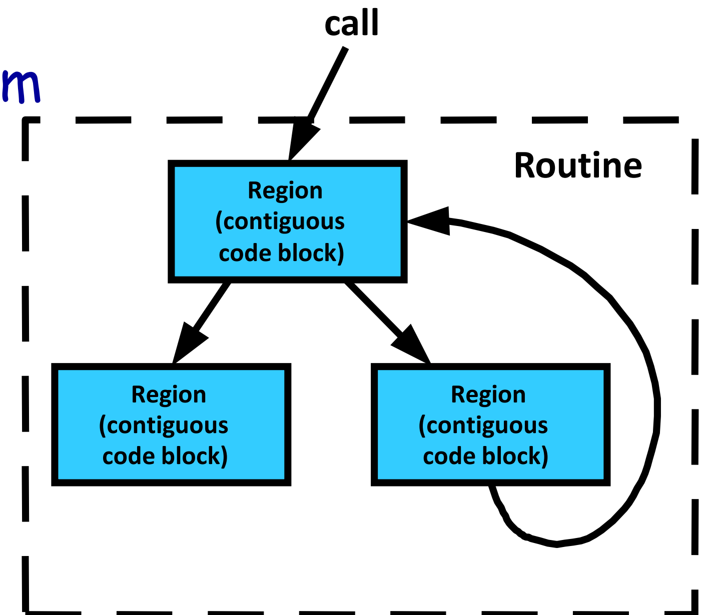
- Part of interpretation process
- Collects:
  - Call target addresses
  - <Source address, Target address> pairs for indirect control transfers
  - …
- Hash table
  - Used for collecting profile data
  - Hash table also used to detect already translated code
    - <x86 address, translation address pairs> are in table

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 a collection of reachable regions
  - Many routines have a single region

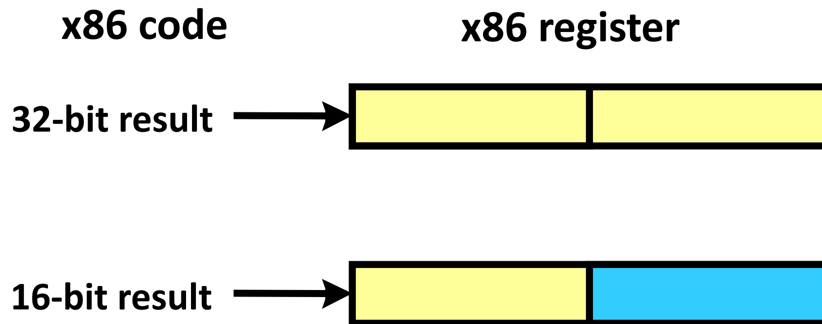


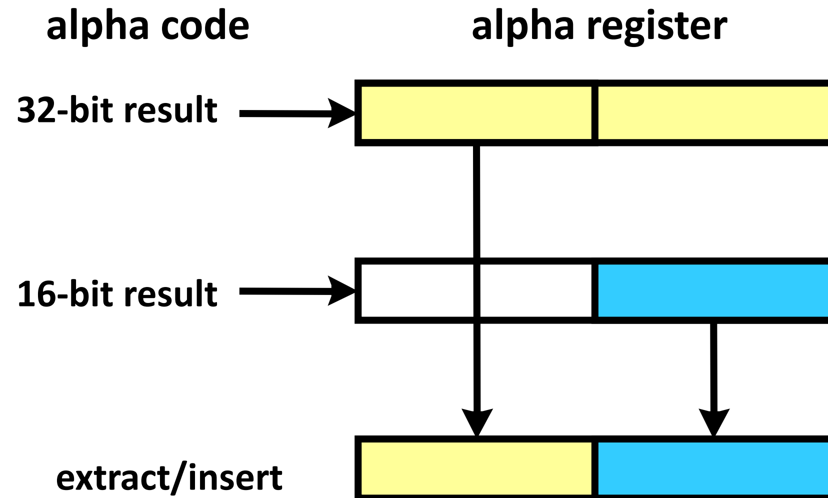
Translator: Build

- 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 allows 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

Translator: Code Selector, Scheduler, Assembler

- Transforms IR to alpha code
- Each x86 instruction is mapped to one or more alpha instructions.
- 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

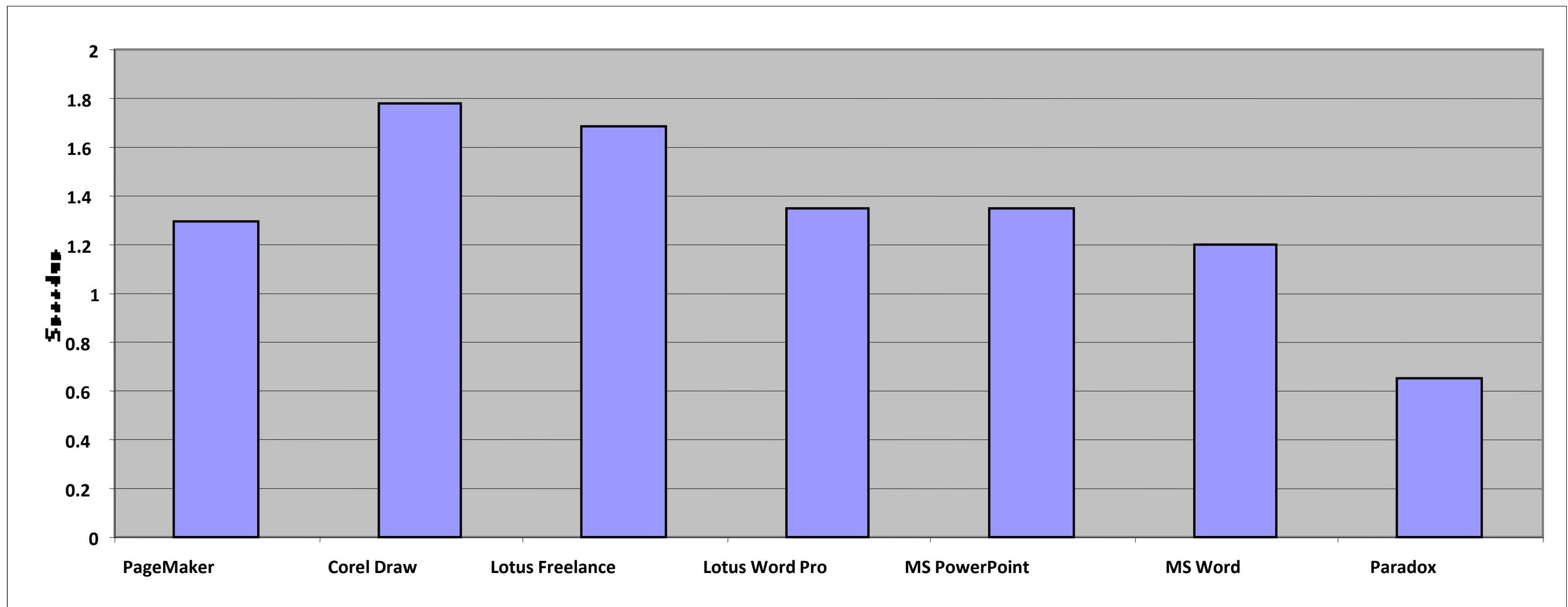
- x86 uses 80 bit floating point for intermediate values
  - Alpha (and most other machines) use 64
  - Emulator uses only 64
    - Good performance
    - Very few applications rely on full 80 bits
- X86 uses stack; sometimes in strange ways
  - Interpreter optimized for strict stack behavior

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
- 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

- Goal: same as high-end x86
- Goal achieved for many integer benchmark
- Floating point – 30% slower
- 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
  - 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
  - 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

Translator, contd.

- X86 peephole optimization
- 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

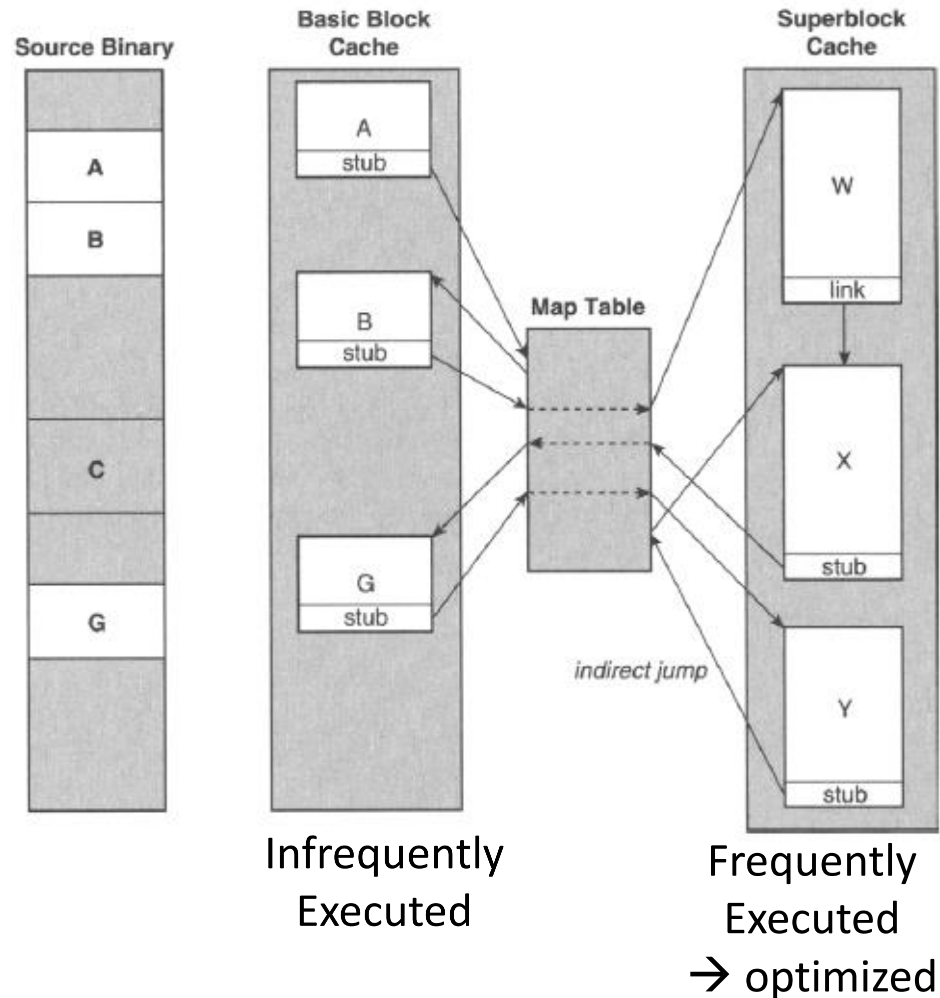
Dynamo
&
Dynamo RIO

http://www.dynamorio.org/

Same ISA Optimization

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

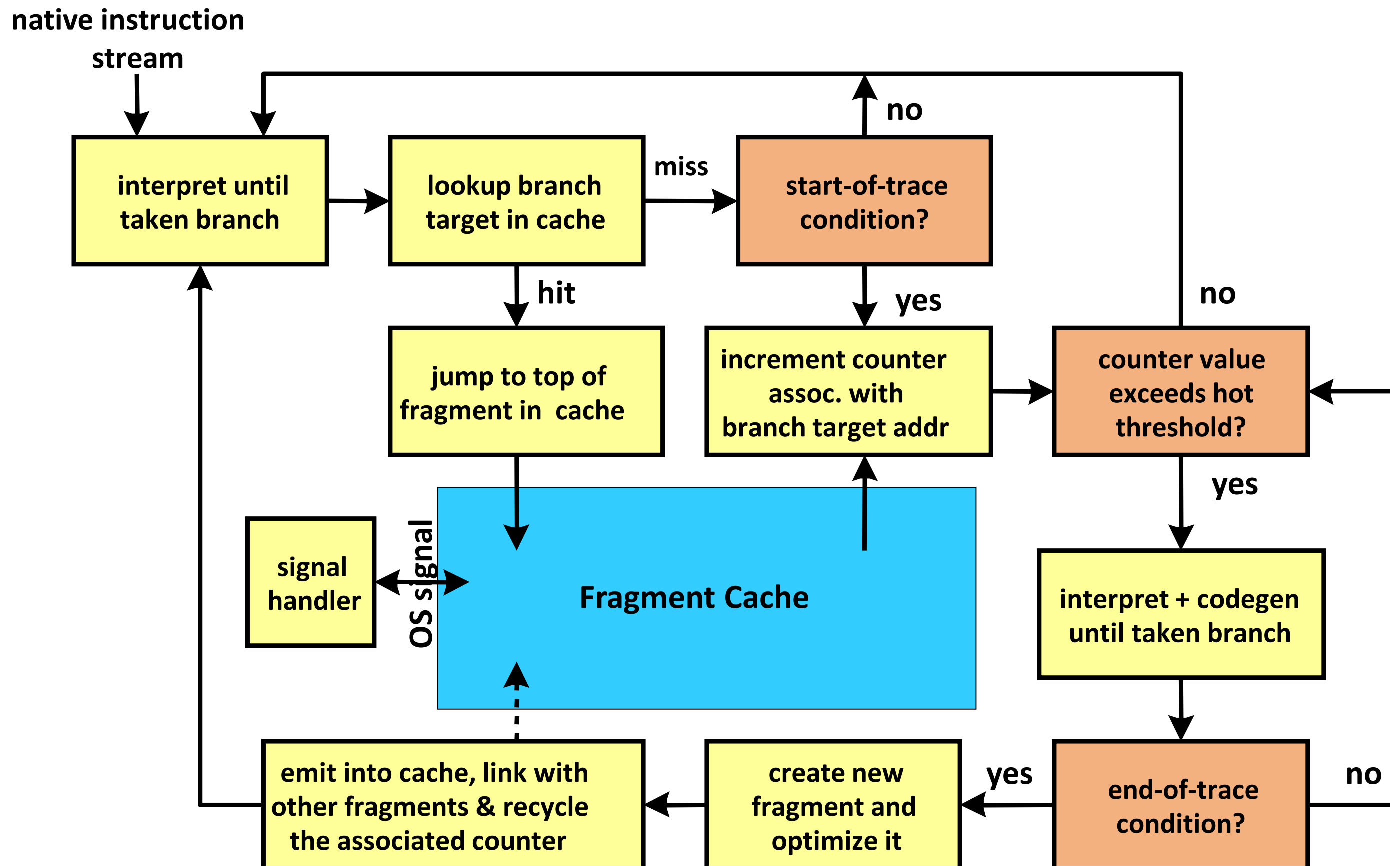


Same-ISA Optimization

- Code patching can be used
  - Replace original code with branches into code cache (saves code some code duplication)
- Can bail-out if performance is lost

Dynamo

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



Basic Operation

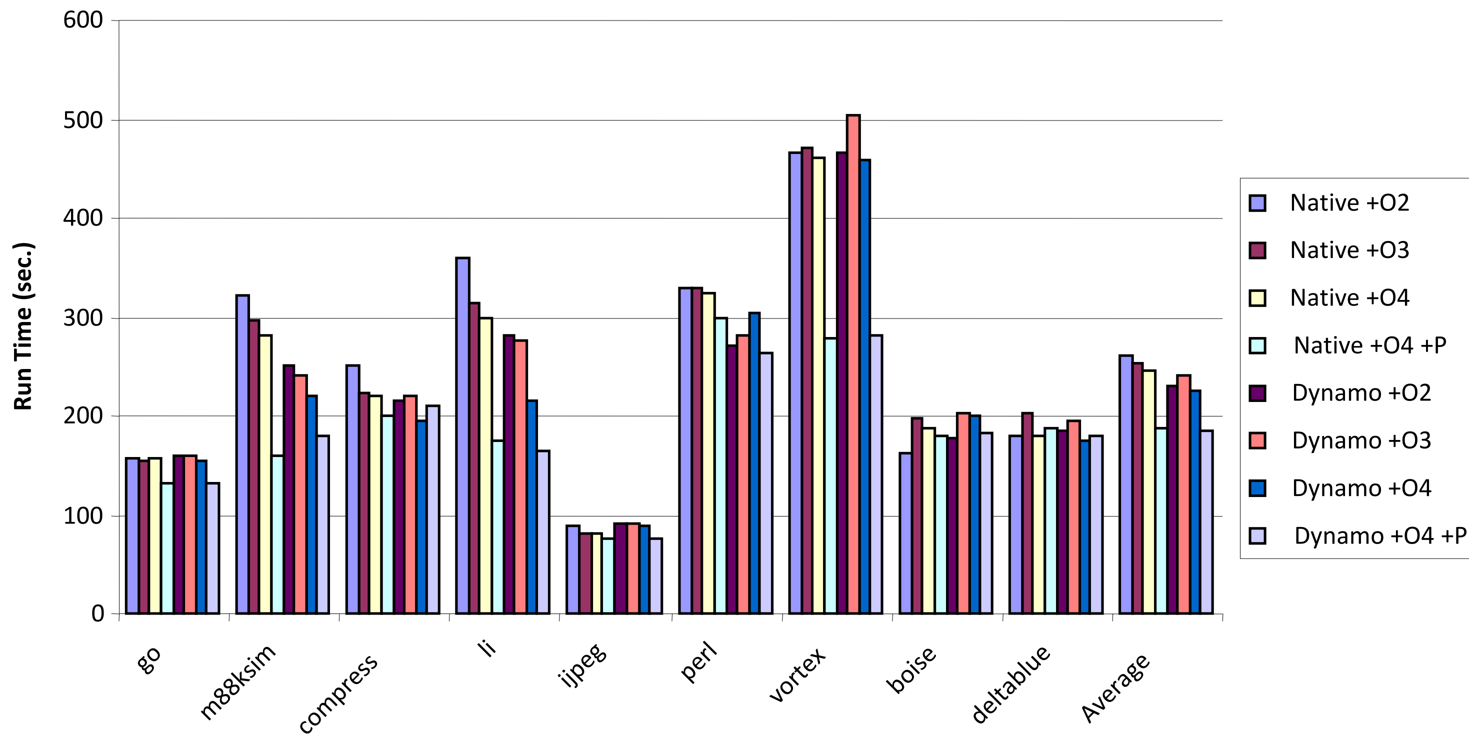
- Staged optimization
  - Beginning with interpretation
  - Then optimized translations
- Translation unit: Fragment
  - Dynamic superblock; can contain calls/returns
- Avg. overhead less than 1.5% of execution time

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