A Unified View of Virtual Machines

First ACM/USENIX Conference on
Virtual Execution Environments

J. E. Smith

June 2005

THE UNIVERSITY of WISCONSIN MADISON

Introduction

Why are virtual machines interesting?

They allow transcending of interfaces
(which often seem to be an obstacle to innovation)

They enable innovation in flexible, adaptive software & hardware, security, network computing (and others)

They involve computer architecture in a pure sense

Virtualization will be a key part of future computer systems

A fourth major discipline? (with HW, System SW, Application SW)

“Oh, the *other kind* of virtual machine”

IBM VM370

VMware

Pascal Pcode

Java

Microsoft CLI

Transmeta Code Morphing

FX!32

Dynamo

Taking a Unified View

“The subjects of virtual machines and emulators have been treated as entirely separate. … they have much in common. Not only do the usual implementations have many shared characteristics, but this commonality extends to the theoretical concepts on which they are based”

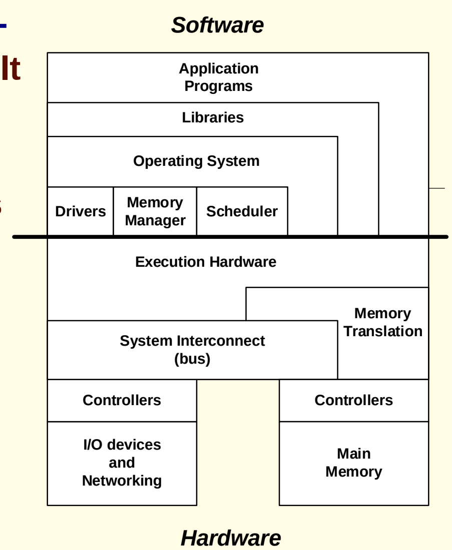
-- Efrem G. Wallach, 1973

Overview

- Virtualization
- A common framework/set of terms
  - The architecture of Virtual Machines
- Higher level concepts
  - There is more than a bag of tricks
- A general set of problems
  - Across VM types
- VM primitives
  - To support solutions to VM problems
- A Killer App
- An academic discipline
  - A course in VMs?

Abstraction

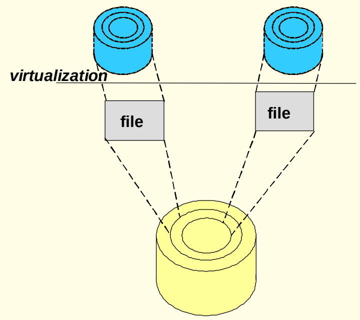
- Computer systems are built on levels of abstraction
- Higher level of abstraction hide details at lower levels
- Example: files are an abstraction of a disk



Virtualization

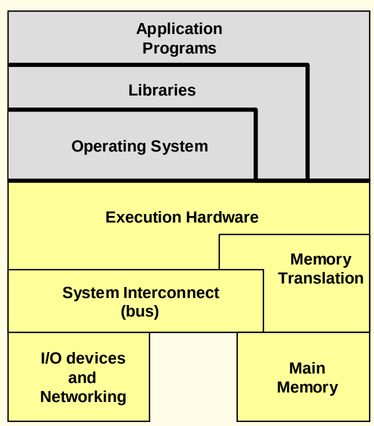
- Similar to abstraction

  Except
  - Details not necessarily hidden
- Construct Virtual Disks
  - As files on a larger disk
  - Map state
  - Implement functions
- VMs: do the same thing with the whole "machine"



The "Machine"

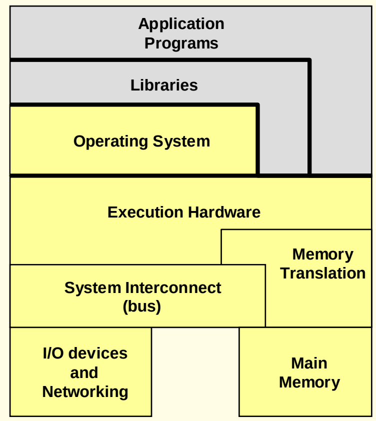
- Different perspectives on what the *Machine* is:
- OS developer



Instruction Set Architecture
- ISA
- Major division between hardware and software

The "Machine"

- Different perspectives on what the *Machine* is:
- Compiler developer

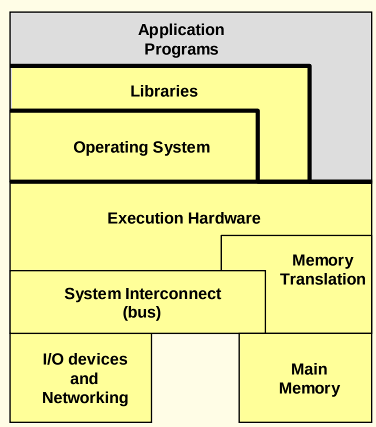


Application Binary Interface

- ABI
- User ISA + OS calls

The "Machine"

- Different perspectives on what the *Machine* is:
- Application programmer

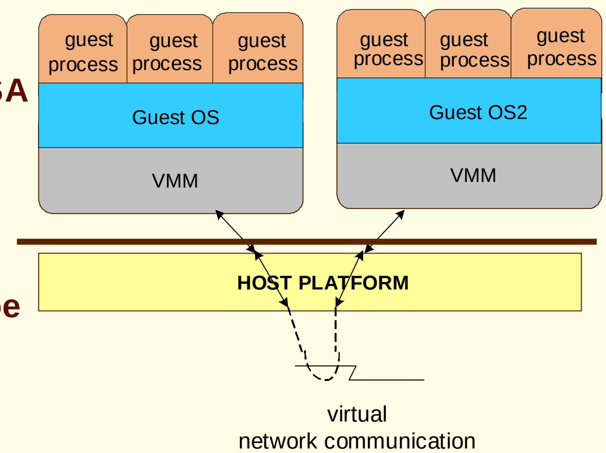


Application Program Interface

- API
- User ISA + library calls

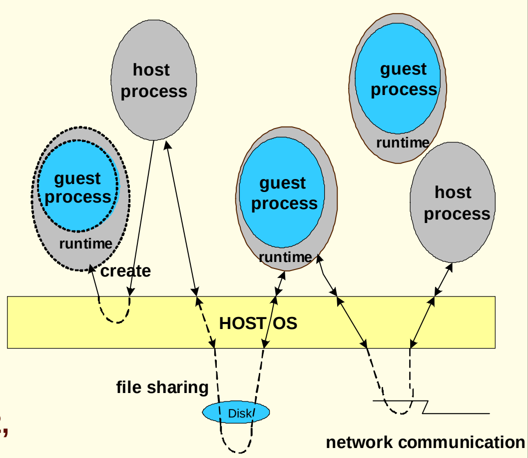
System Virtual Machines

- Provide a system environment
- Constructed at ISA level
- Persistent
- Examples: IBM VM/360, VMware, Transmeta Crusoe



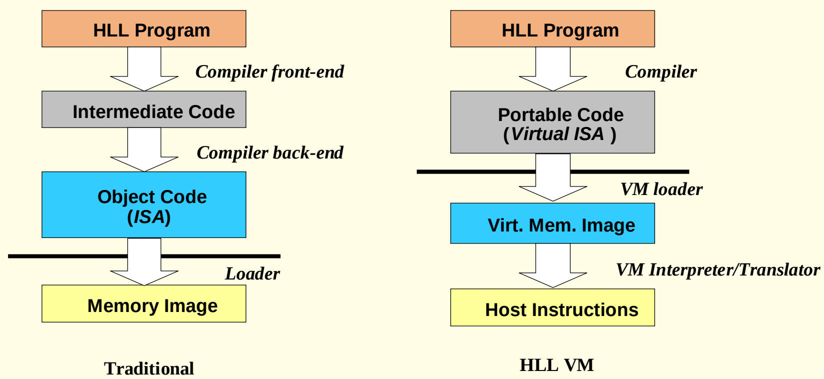
Process Virtual Machines

- Constructed at ABI level
- *Runtime* manages guest process
- Not persistent
- Guest processes may intermingle with host processes
- As a practical matter, guest and host OSes are often the same
- Dynamic optimizers are a special case
- Examples: IA-32 EL, FX!32, Dynamo

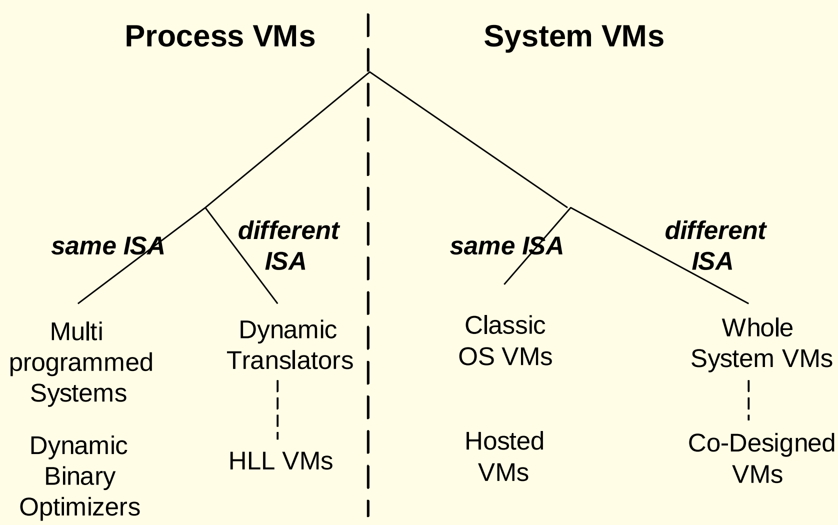


High Level Language Virtual Machines

- Raise the level of abstraction
  - User higher level virtual ISA
  - OS abstracted as standard libraries
- Process VM (or API VM)



The Virtual Machine Space



A Discipline or a Bag of Tricks?

- A number of VM techniques were invented (and re-invented) on an *ad hoc* basis
  - Many common emulation techniques:
    Code caches, Inline caching, Staged emulation…
- Is there more than a collection of techniques?
  - I.e., Some over-arching problems and "meta-architecture" solutions?
  - Guidelines for defining new architectures
    to make them efficiently virtualizable
  - Useful primitives for supporting VMs

Virtualizability (Goldberg, Popek, '74)

- Classic work in formalizing OS VM concepts
- Defines basic VM properties
- Defines properties of instruction sets
- Proves that VMM can be constructed if instruction set properties hold
- Extends to recursive VMs

Privileged Instructions, Definition:

- ***Trap if executed in user mode; not in supervisor mode***
- **This is a defined term**
- **Privileged instructions are *required* to trap**
  - No-op in user mode is not enough

Control Sensitive instructions:

Instructions that provide control of resources:

1. **All instructions that change the amount of (memory) resources (or the mapping)**
2. **All instructions that change the processor mode**

   Examples:
   - Load TLB (if TLB is architected)
   - Load control register
   - Return to user mode

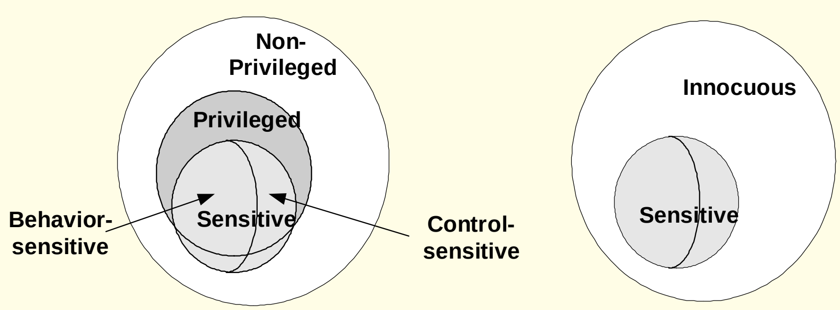
Behavior Sensitive instructions:

Instructions whose behavior depends on configuration of specific resources:

1. **All instructions whose results depend on the mapping of physical memory**
2. **All instructions whose behavior depends on the mode**

   Examples:
   - Load physical address
   - POPF (Intel x86): Interrupt-enable flag remains unaffected in user mode

Instruction Types -- Summary



- **Innocuous Instructions:** Those that are not control or behavior sensitive

Virtual Machine "requirements"

1. **All innocuous instructions are executed by the hardware directly**
2. **The allocator must be invoked when any program attempts to affect system resources**
3. **Any program executes exactly as on real hardware except**
   - For timing
   - Availability of system resources

- **A VMM satisfies all three requirements**
  - Perhaps better to say "efficient" VMM

Virtual Machines: Main Theorem

A virtual machine monitor can be constructed if the set of sensitive instructions is a subset of the set of privileged instructions

Proof shows

Equivalence by interpreting privileged instructions and executing remaining instructions natively

Resource control by having all instructions that change resources trap to the VMM

Efficiency by executing all non-privileged instructions directly on hardware

A key aspect of the theorem is that it is easy to check

Recursive Virtualization

Running a VMM as a VM on a VM on a VM….

***Theorem*: A conventional third generation computer is recursively virtualizable if it is (a) virtualizable, and (b) a VMM without any timing dependences can be constructed for it**

***Proof* – A VMM is a program and from the VM theorem will be "identically performing" except for timing dependences and resource constraints.**

Timing is excluded in the theorem;

Resource constraints only limit the depth of recursion.

Designing-In Virtualizability

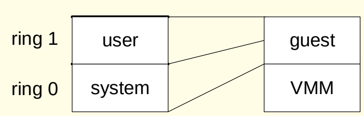
- **Goldberg and Popek work is in this direction**
- **HLL VMs (Java and CLI) are examples**
- **Problems with conventional ISAs**
  - protecting VM software
  - ISAs with fixed logical resources
  - state recoverability and mobility
  - HLL VMs solve, sidestep, or avoid these issues

VM Architecture: Protecting VM Software

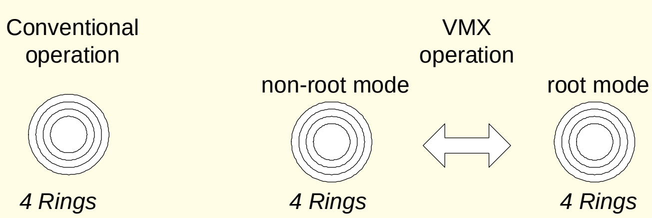
- VM SW must be close enough to conventional SW to interact efficiently
- But, VM implementation SW must be protected from conventional SW (and must not change behavior of conventional SW)

Protecting VM Software: System VMs

- Use Ring Compression
  - Leads to inefficiency in ring emulation



- Intel VT-x solution – add another set of rings

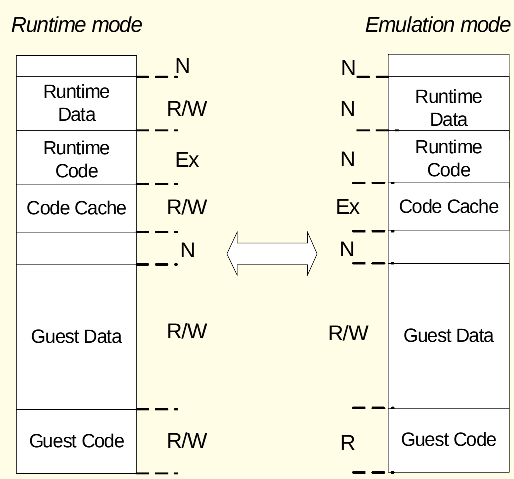


Protecting VM Software: Process VMs

- Can also use ring compression in a sense
- Both runtime SW and application SW are part of user process' address space
  - Dynamo "mode switch"
    between emulation mode and runtime mode

Protecting VM Software: Process VMs

- Translated code should only access guest memory image
- Translated code should not jump outside code cache (emulation s/w sets up links)
- Change protections on "mode switch"
- Multiple system calls at mode switch time
  - High overhead

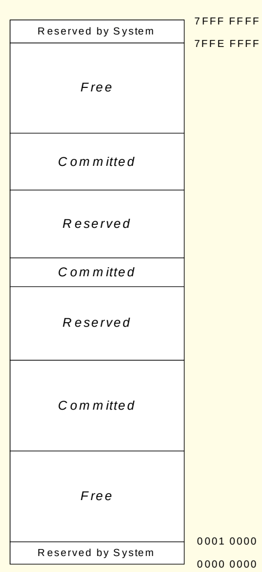


Protecting VM Software: Process VMs

- Needs a good solution
- Performance issues (just noted)
- Problems with access to VM-generated tables in emulation mode
  - Some VM tables remain unprotected
  - Register spill areas

Fixed Logical Resources

- Conventional ISAs support fixed memory and register spaces
- System VMs have mechanisms that virtualize these resources
- Process VMs are more limited
  - VM runtime consumes part of memory space
  - Optimizers need more register resources



Fixed Logical Resources

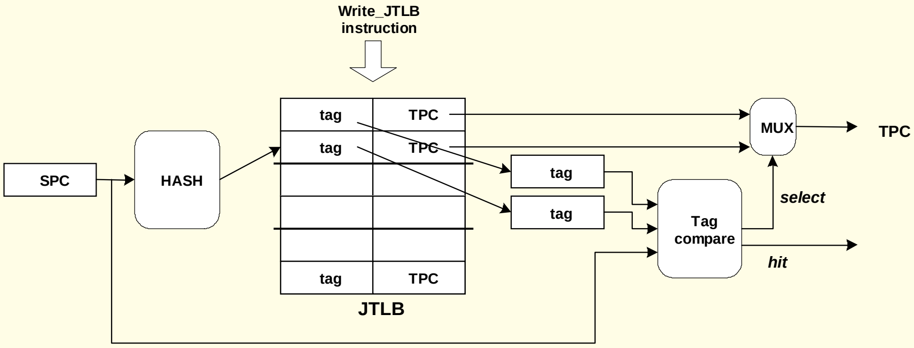
- **No problem for IA32 process VMs on a RISC**
  - RISCs have 64-bit address space and large register file
  - Problem for same-ISA optimizers
  - Will be problem when 64-bit x86 becomes the common SW base
- **HLL VMs have a very interesting approach**
  - Key point is that logical resources do not have a bounded size
  - Can the HLL VM approach be extended to a complete ISA?
  - At some point the logical must become physical

Precise Traps (Exceptions)

- **A problem at every level of the architecture stack**
  - The one problem everyone seems to agree on
- **Actually, precise traps are a *solution* to problems:**
  - Recoverability
  - Capturing State
- **Recoverability**
  - After exception handling, can execution resume correctly?
  - Often constrains optimizations due to problems w/ state update
- **Capturing state**
  - Can entire execution state be picked up and moved?
  - Common assumption for standard processes
  - VM techniques can be applied for full systems
  - HLL VMs do not have this capability – rely on underlying process
- **Are there other solutions?**

Perf. Primitives: Code Cache Support

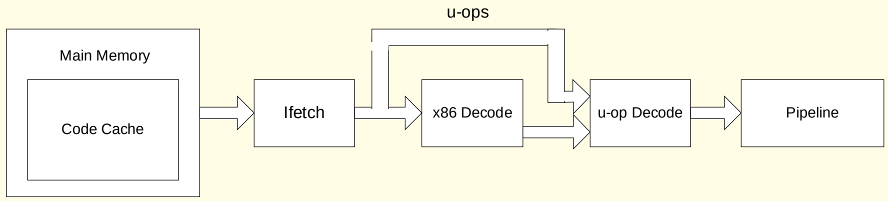
- Indirect jumps are big performance problem
  - Translation of source PC to target PC
- Add Jump TLB
  - A hardware cache of dispatch table entries
  - Similar to software-managed TLB in virtual memory



VEE '05 (c) 2005, J. E. Smith 33

Perf. Primitives: Visible Micro-ops

- Factor out u-ops and let VM software use them
- Example: HW/SW co-designed Java VM
  - Avoid extra translation layer that discards meta-data



VEE '05 (c) 2005, J. E. Smith 34

Problems with Perf. Enhancements

- How can we make them visible without making them legacy?
  - Already Intel has “legacy microcode”
  - Maybe in a vertically integrated environment…
    e.g. IBM AS/400
- Recursive virtualizability
  - What if conventional software uses the feature?
  - Can the software still run on a VM that relies on the feature?
    JTLB
    Intel VT-x

A Killer App?

Security!

- HLL VMs
  - Support for secure network computing is a major objective
  - Security built-in – and continues to evolve
- Process VMs
  - All code can be inspected before being executed
  - Program Shepherding
- System VMs
  - Provide isolation
  - Simple VMM can provide final checks
- Co-Designed VMs
  - Concealed memory allows secure implementation of primitives

A Course in VMs?

- Teaches architecture in the pure sense
  - The meaning and importance of compatibility
- Teaches the “non-obvious”, but difficult parts of an architecture
- Simple virtualization also means simple abstraction => better interface design (?)
- Pulls together virtually all the levels of computer system hardware and software
  - both + and -

Conclusions

- Common framework/terms?
  - Not now – but should be settled on
- More than a bag of tricks?
  - Yes, there are some higher concepts
  - Demonstration through examples
  - Also performance primitives

    but not without problems
- Is it an academic discipline?
  - A course in VMs can be done, but it is challenging
- A unifying conference?
  - VEE !!