

Mechanisms and constructs for System Virtualization

Content Outline

- Design goals for virtualization
- General Constructs for virtualization
- Virtualization for:
 - System VMs
 - Process VMs
- Prevalent trends: Pros and cons

Requirements for virtualization

- Will depend on what kind of virtualization one is attempting
 - System Virtual Machines
 - Process Virtual Machines

Goals of System Virtualization

- Virtualization goals: (Goldberg(1974) first survey paper on virtualization technologies.)
 - Efficiency: Virtual Machine must be as efficient as the non-virtualized machine, if not more!
 - Isolation: Virtual Machine is an isolated duplicate of the real machine.
 - Safety: Virtual Machine can access resources explicitly allocated to it only.

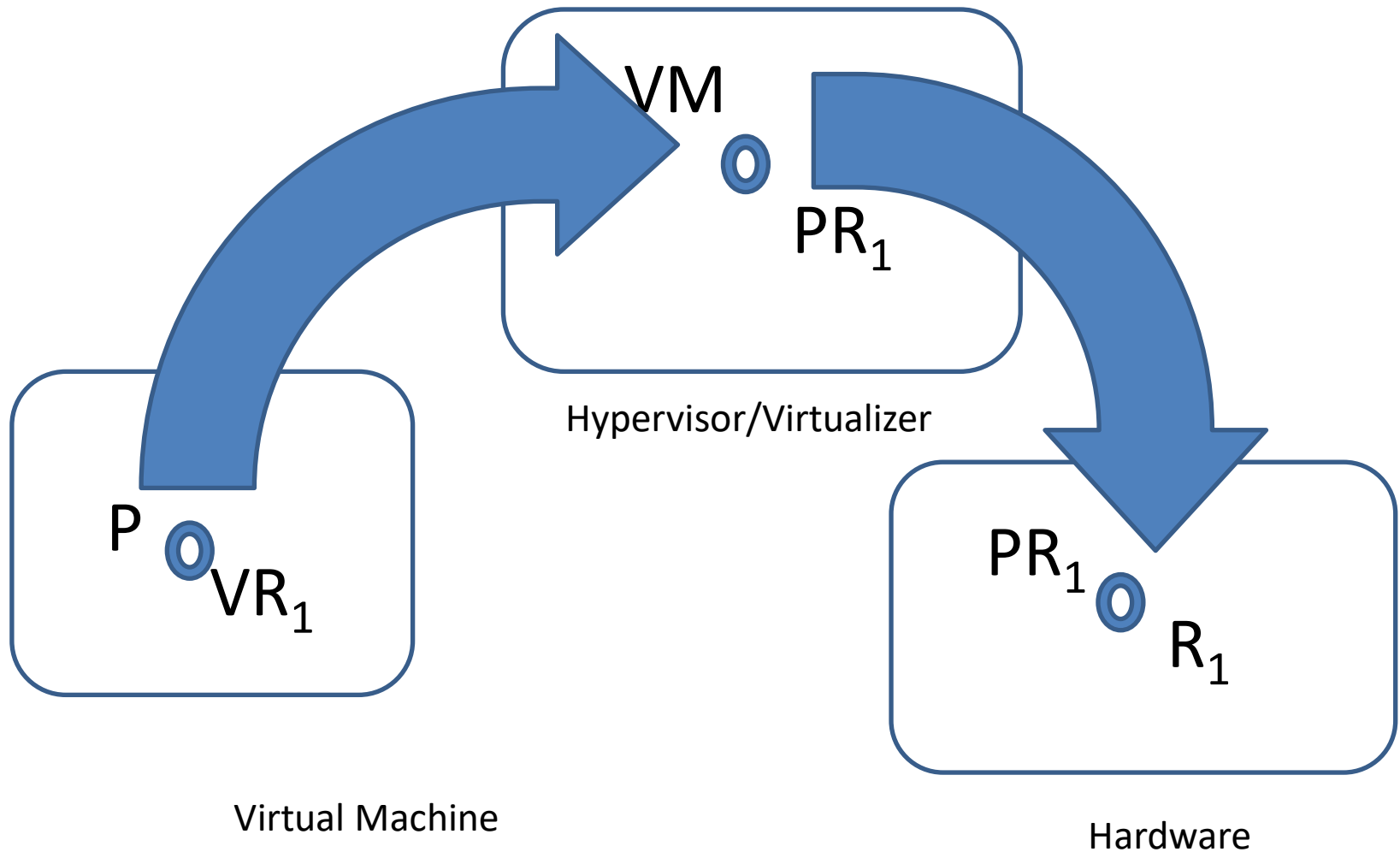
Hardware Virtualizer - Desirables

- Need for hardware virtualizer that supports:
 - System hygiene: Virtual machines should not be built using instruction trap and emulation method since such approaches are clumsy and awkward.
 - Software simplicity: Inherent hardware support for virtualization would make the VMM a small and simple program that further contributes to system reliability and security.
 - System performance: Systems that are designed to support virtualization should operate more efficiently than their non-virtualized counterparts.

f-map virtual resource definition

- The f-map transforms the virtual resource name to its corresponding real resource name.
- f-map is transparent to the VM in its state of execution in privileged or non-privileged mode. Hence, the VM executes as it would on a real machine.
- All real and virtual resources are under the control of the VMM. VMM manipulates and invokes the f-map, and any f-map violation passes the control to the VMM.

Conceptual view of f-map

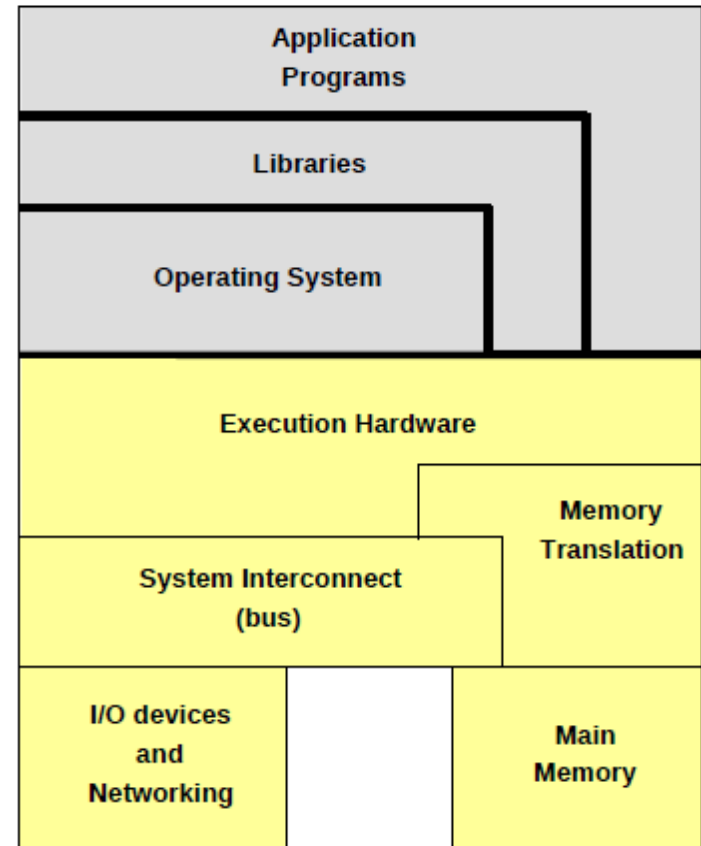


Requirements for System VMs

- Support for multiple ISAs
 - Static/dynamic Instruction translation/interpretation
- Support for multiple processor privilege modes
 - ISA extended to support use of selective privileges across these different privilege modes
- Memory support for multi-level address translation
- I/O device support for virtualization
 - I/O device virtualization support for multiple VM access and associated address translations.

HAL/ISA Interface

- Hardware Abstraction Layer (HAL) & ISA form the basic interface between hardware and any software.
- Mostly used by OS kernel programmers
- Accessible to users through
 - User ISA + OS Systems Calls via the libraries and APIs
- System virtual machine realization



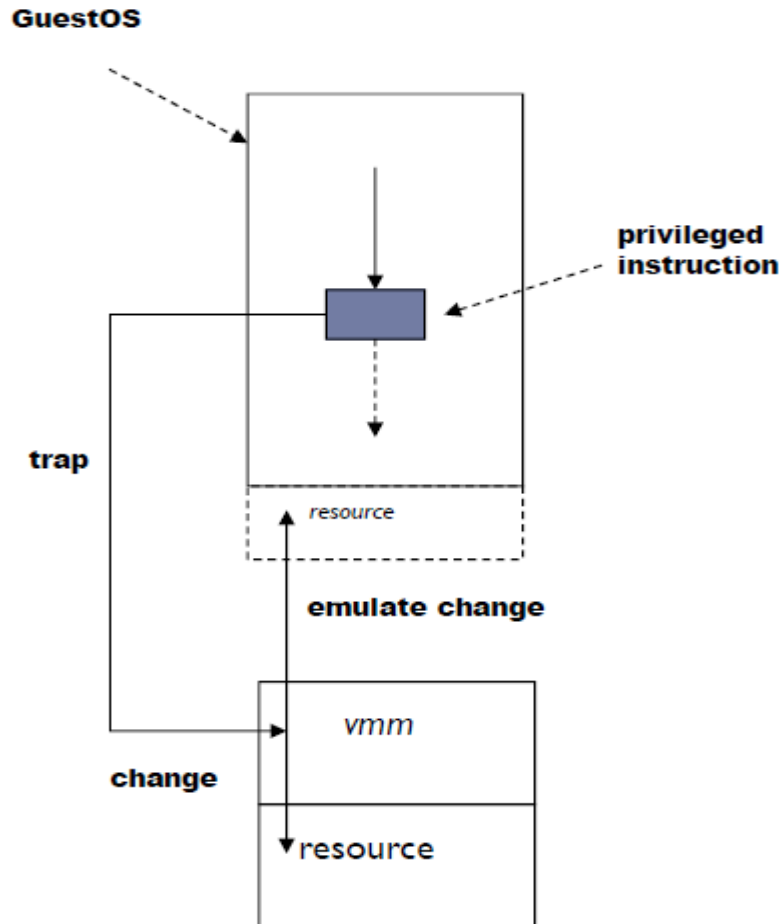
Mechanisms for System Virtualization

- Emulation
- Para-virtualization
- Hardware-virtualization

What needs to be virtualized - All aspects of the ISA

- User + System ISA
- Register Set
- Memory addressing architecture
- Traps & Interrupts architecture
- I/O Devices
- BIOS/Boot Loader

Emulation use in Virtualization



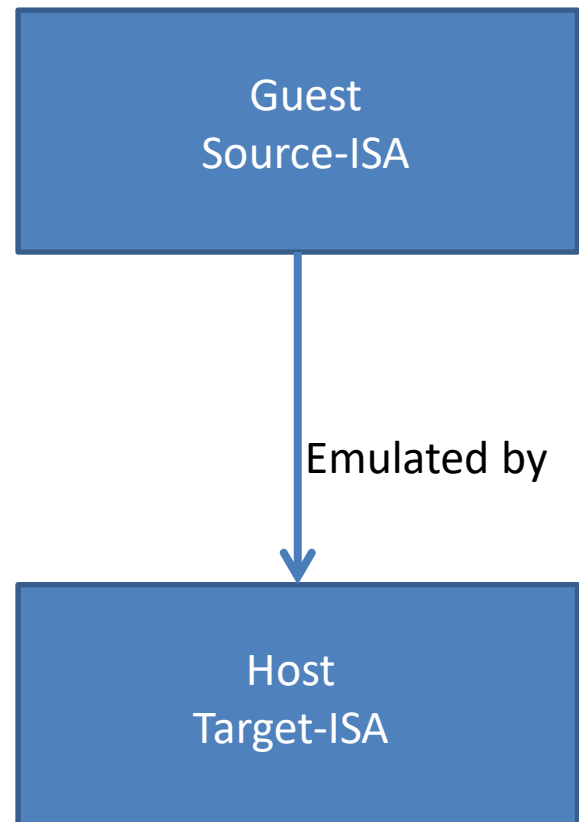
- GuestOS unaware of executing on virtual resources and in de-privileged mode
- Privileged instructions executed by GuestOS trap to VMM
- VMM emulates the effect of execution of privileged instructions on physical resources to GuestOS

Emulation for ISA virtualization

- What is emulation?
- Need for emulation
- Mechanisms for emulation
 - Interpretation
 - Binary Translation

Emulation

- Mechanism to implement the interface and functionality of one system on a system that has a different interface and functionality.
- Why would one use emulation?
 - Validation of new ISA
 - Binary code execution on varied runtimes



Advantages & Pitfalls to using Emulation

- Support for legacy runtime environments
 - No code changes (emulator takes care of translation at runtime)
 - Existing applications get benefit of improved system architecture
 - Process VMs are native models for emulation based virtualization
- System VMs need some support from hardware
 - More than two privilege rings
 - ISA should have all privileged instructions to trap or raise exceptions such that GuestOS privileged instructions can transit to VMM for correct execution.

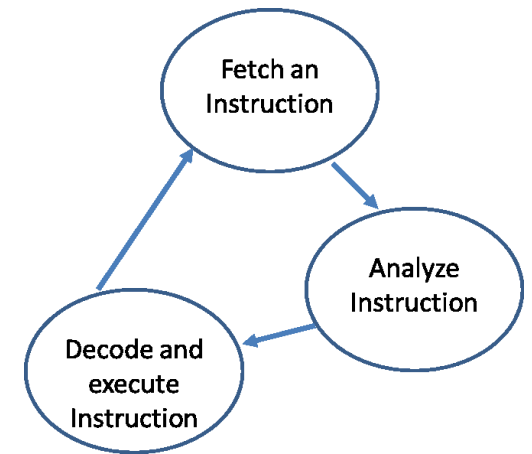
Anatomy of ISA

- ISA defines a set of instructions that are used to carry out two distinct functions on the system hardware:
 - Use the system for computing (User-ISA)
 - Manage hardware resources (System-ISA)
- Emulation is mostly concerned with the User-ISA part for Process-VMs. However, System-VMs may also need emulation of System-ISA or extend the System-ISA.

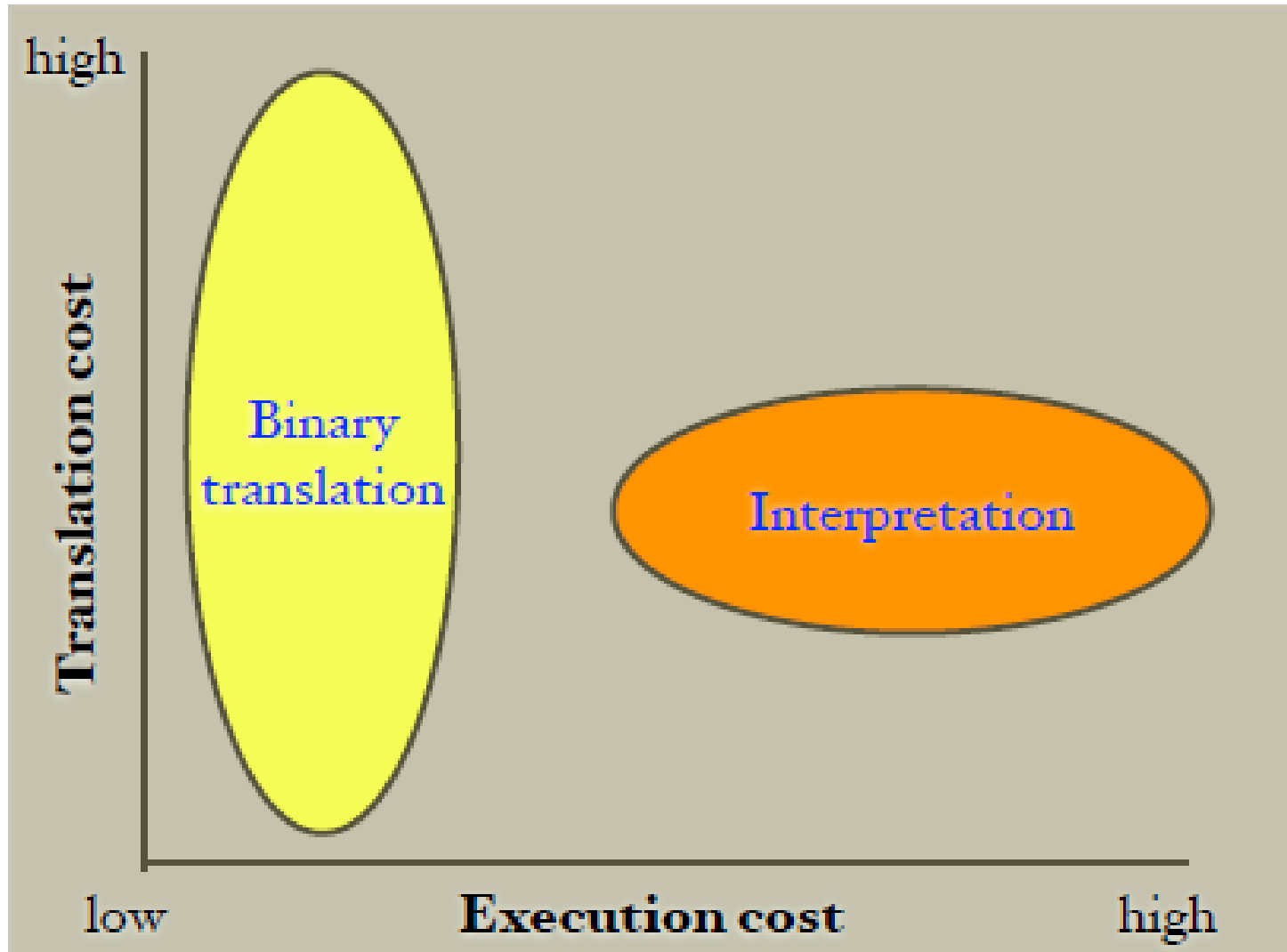
Reference: Appendix-A Real Machines from Smith and Nair's book on Virtual Machines

Techniques for Emulation

- Interpretation: Translate instruction by instruction of the source binary and execute
 - Basic interpretation
 - Threaded Interpretation
 - Precoding
 - Direct threaded interpretation
 - CISC Interpretation
- Binary Translation: Translate a block of source instructions to target instructions, additionally save the translated code for repeated execution
 - Precoding
 - Dynamic Translation



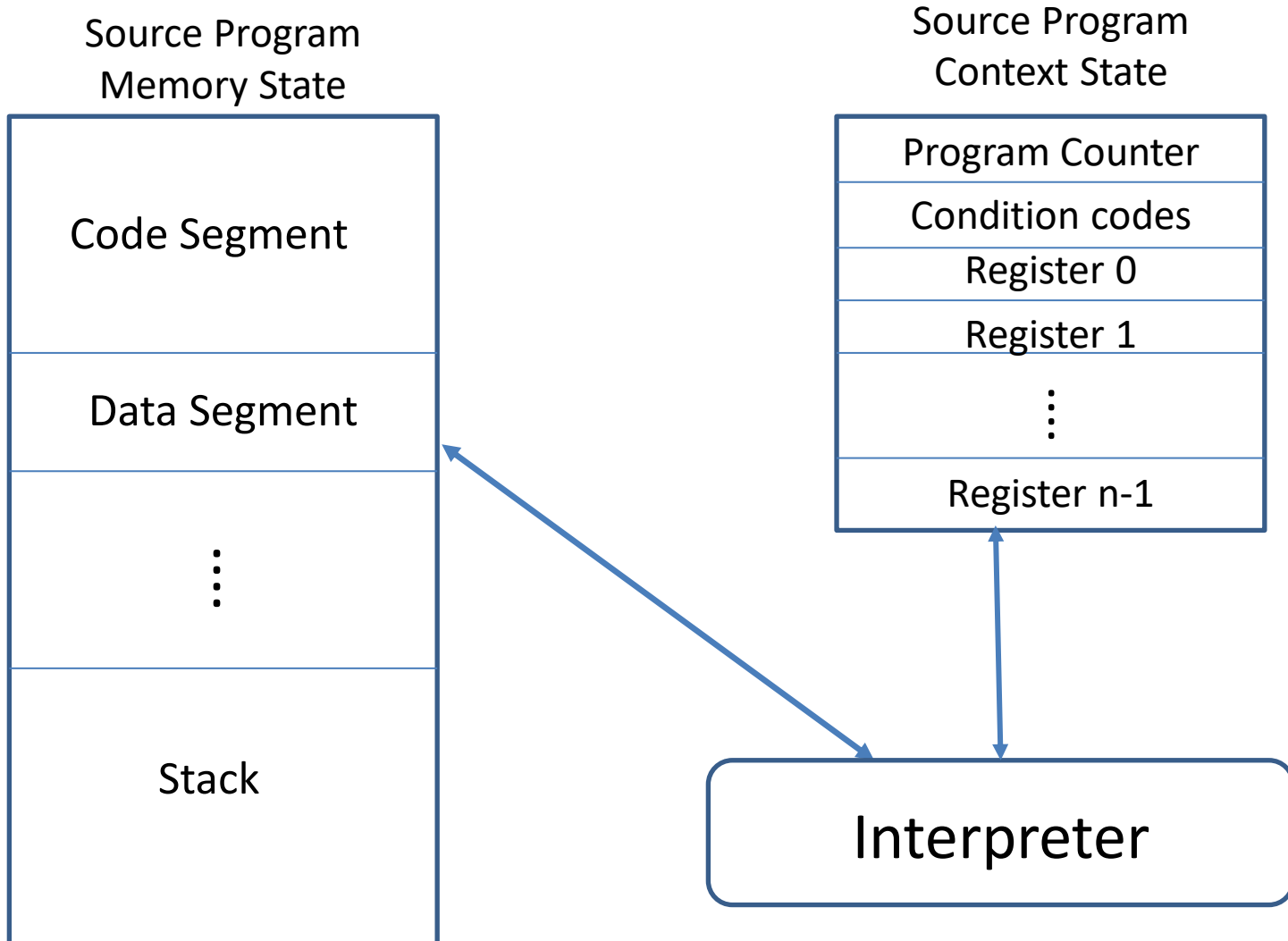
Forms of Emulation



Basic Interpretation

- Goal: Apply interpretation techniques to program binaries.
- Definition: “An interpreter program emulates and operates on the complete architected state of the machine implementing the source ISA, including all architected registers and main memory”

Interpreter Overview



Basic Interpreter: Decode & Dispatch

```
while ( !halt && !interrupt) {  
    inst = code[PC];  
    opcode = extract(inst, 31,6);  
    switch(opcode) {  
        case LoadWordAndZero:  
            LoadWordAndZero(inst);  
        case ALU: ALU(inst);  
        case Branch: Branch(inst);  
        :  
    }  
}
```

- Steps through source program instructions and decodes and modifies the state based on the instruction.
- Decoded instruction is dispatched to the interpreter routine to execute on the target ISA.
- One source instruction can translate to many target instructions.
- Many branch instructions may not be pipeline friendly (conditional branch).
- Basic interpretation has high performance overheads.

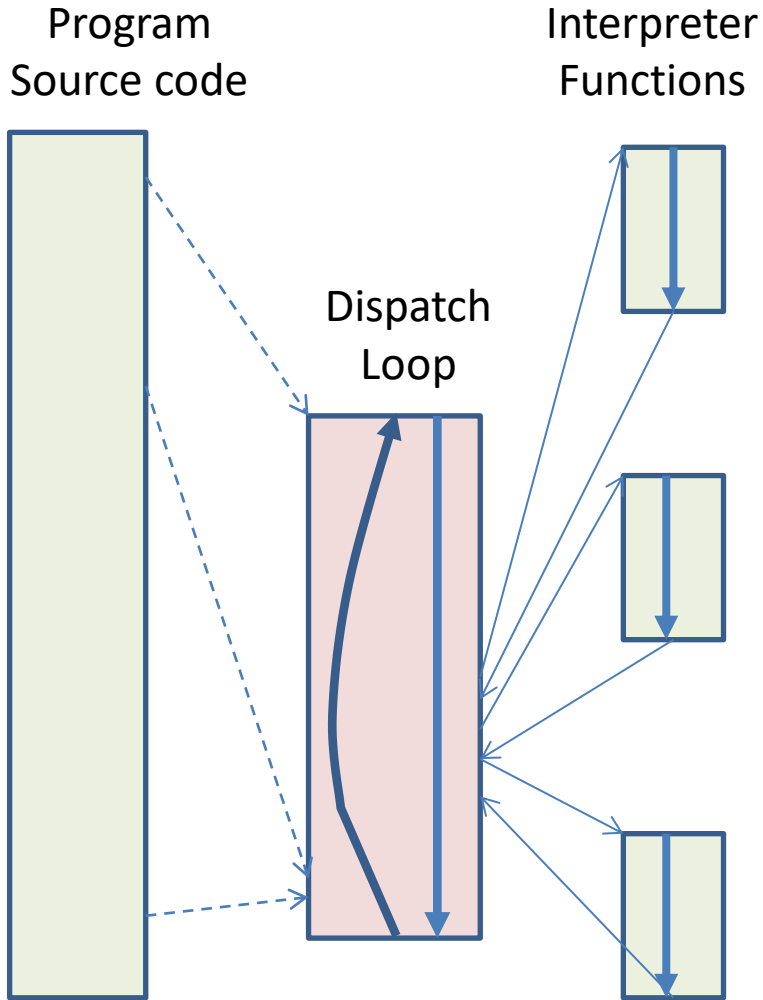
Threaded Interpretation

- In threaded interpretation, the dispatch function is added to each of the decoding function.
- This eliminates some overhead, namely
 - Branch to case switch on the dispatch loop
 - Branch to return from interpreter routine
 - Branch to exit the dispatch loop.

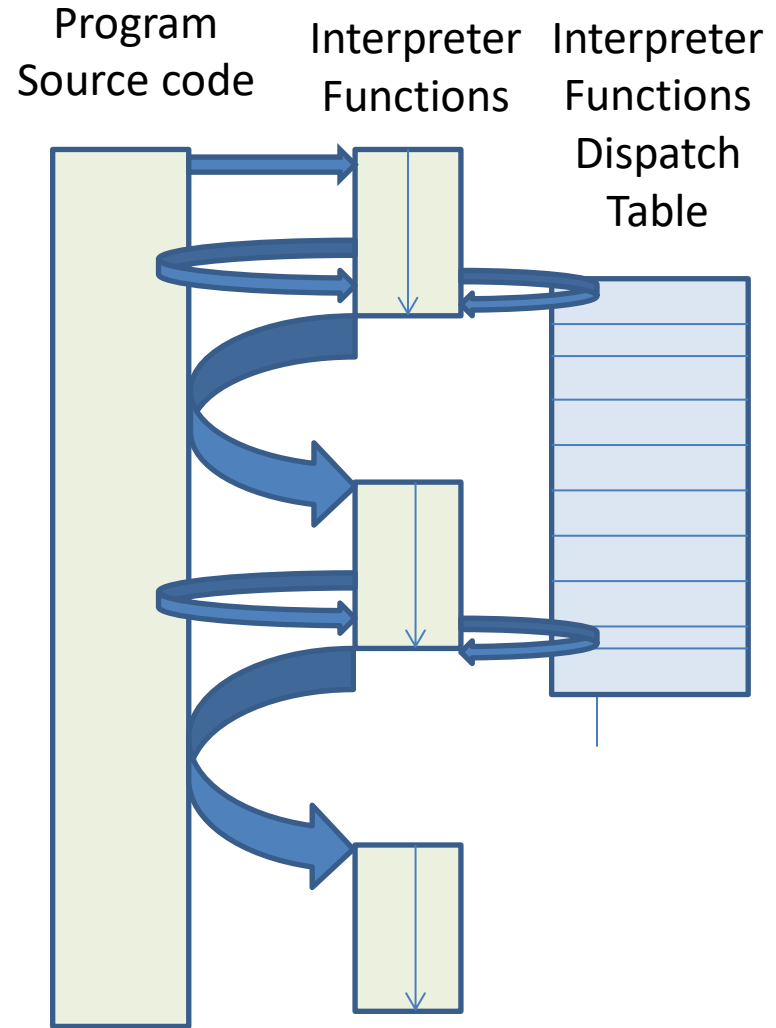
Instruction function list

```
LoadWordAndZero(inst){
    RT = extract(inst, 25,5);
    RA = extract(inst,20,5);
    displacement = extract(inst, 15, 16);
    if (RA==0) source = 0;
    else source = regs[RA];
    address = source + displacement;
    regs[RT] = (data[address]<<32 >>32);
    PC = PC + 4;
    If (halt || interrupt) goto exit;
    inst = code[PC];
    opcode = extract(inst,31,6);
    extended_opcode = extract(inst,10,10);
    routine = dispatch[opcode,extended_opcode];
    goto *routine;
}
```

Interpretation Methods



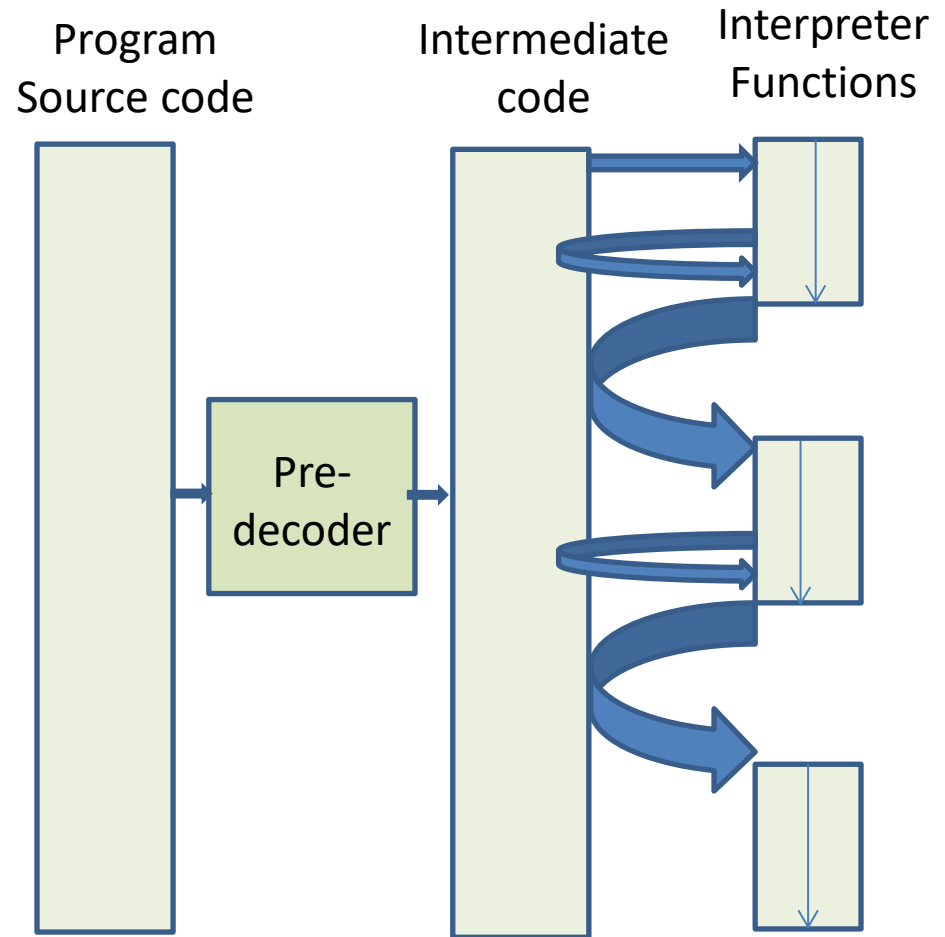
Basic Decode and Dispatch Method



Indirect Threaded Interpretation

Precoding & direct threaded Interpretation

- Instruction specific translation is pre-decoded and stored.
- Pre-decoding allows for direct threaded interpretation
- Overheads on lookup table is eliminated.



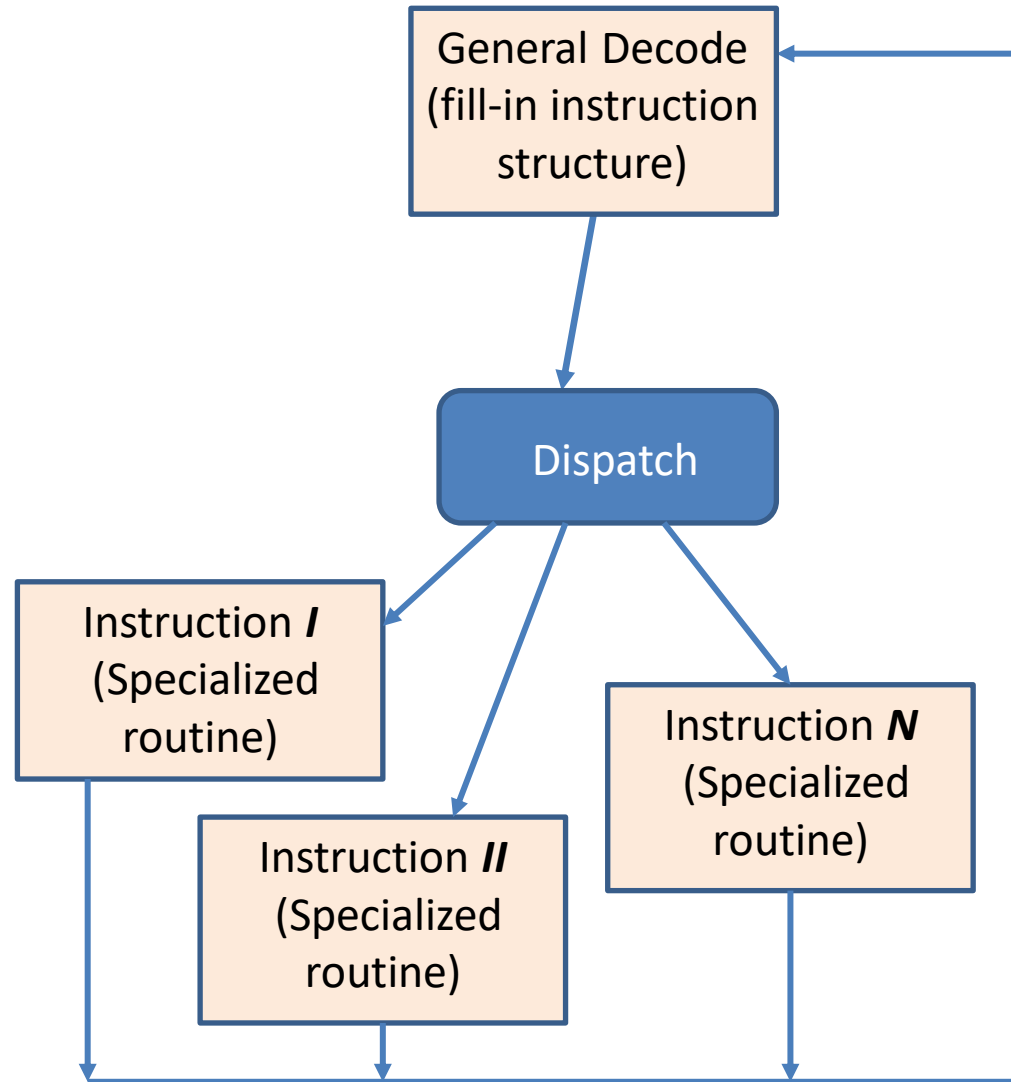
Direct Threaded Interpretation

Emulation by Interpretation Summary

- Basic interpretation uses decode-and-dispatch, which is simple but causes many branches.
- Indirect threaded interpretation eliminates the dispatch loop but uses dispatch table
- Precoding techniques amortize on elimination of redundant extractions for repeated instructions.
- Direct threaded interpretation uses precoding and eliminates dispatch table.

CISC Interpretation*

- CISC versus RISC:
 - RISC ISAs have regular instruction formats
 - CISC ISAs have variety of instruction formats with variable instruction lengths and fields
- General approach to interpreting a CISC ISA
 - Interpretation is divided into two phases:
 - Scan and decode various instruction fields
 - Dispatch for jumping to specialized routines for each instruction type
- Generic interpretation is optimized for faster execution using common shared routines and threaded approaches.



Binary Translation

- Translate source binary program to target binary before execution
 - Pre-decode
 - Enables optimization on the *native* code
 - Handles jumps by elimination
 - Better performance compared to interpretation
 - Source program state mapping onto the host or target program state(*state mapping*) requirement

Binary Translation

Basic Idea

x86 Source Binary

```
addl %edx,4(%eax)
movl 4(%eax),%edx
add %eax,4
```

Translate to PowerPC Target

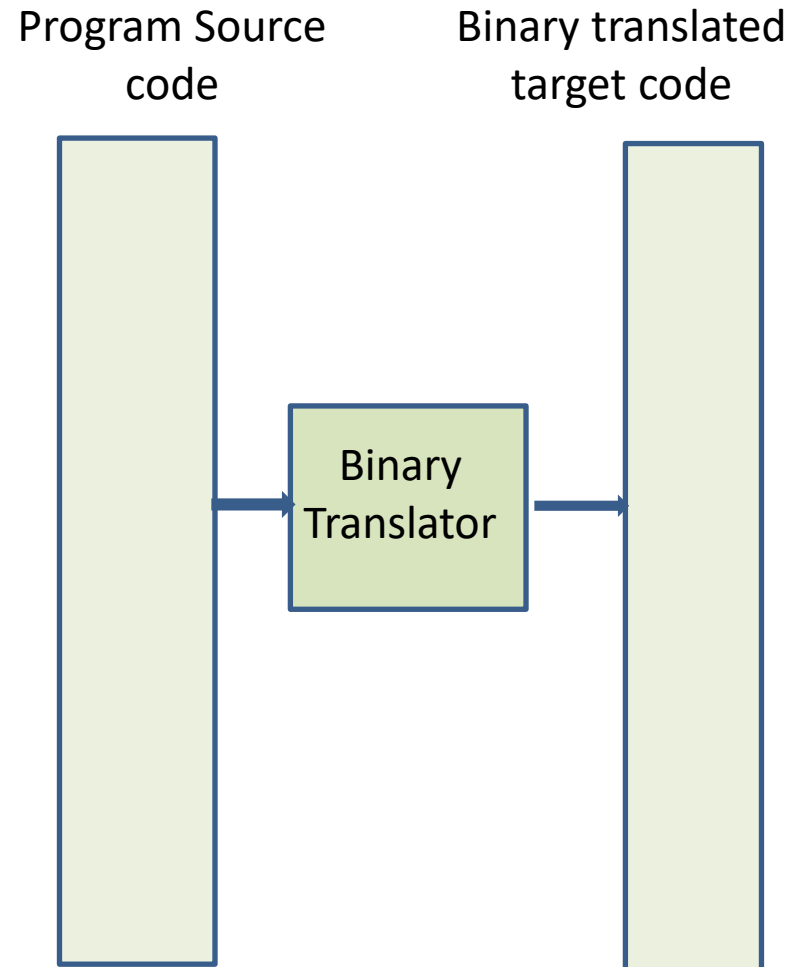
r1 points to x86 register context block
r2 points to x86 memory image
r3 contains x86 ISA PC value

Example Code Sample

```
lwz   r4,0(r1) ;load %eax from register block
addi  r5,r4,4  ;add 4 to %eax
lwzx  r5,r2,r5 ;load operand from memory
lwz   r4,12(r1);load %edx from register block
add   r5,r4,r5 ;perform add
stw   r5,12(r1);put result into %edx
addi  r3,r3,3  ;update PC (3 bytes)
lwz   r4,0(r1) ;load %eax from register block
addi  r5,r4,4  ;add 4 to %eax
lwz   r4,12(r1);load %edx from register block
stwx  r4,r2,r5 ;store %edx value into memory
addi  r3,r3,3  ;update PC (3 bytes)
lwz   r4,0(r1) ;load %eax from register block
addi  r4,r4,4  ;add immediate
stw   r4,0(r1) ;place result back into %eax
addi  r3,r3,3  ;update PC (3 bytes)
```

Static Binary Translation: Precoding

- Binary translation refers to the mechanism wherein the source binary code is translated to the target binary code.
- Each source instruction translation is customized and state register mapping from source to target is employed.

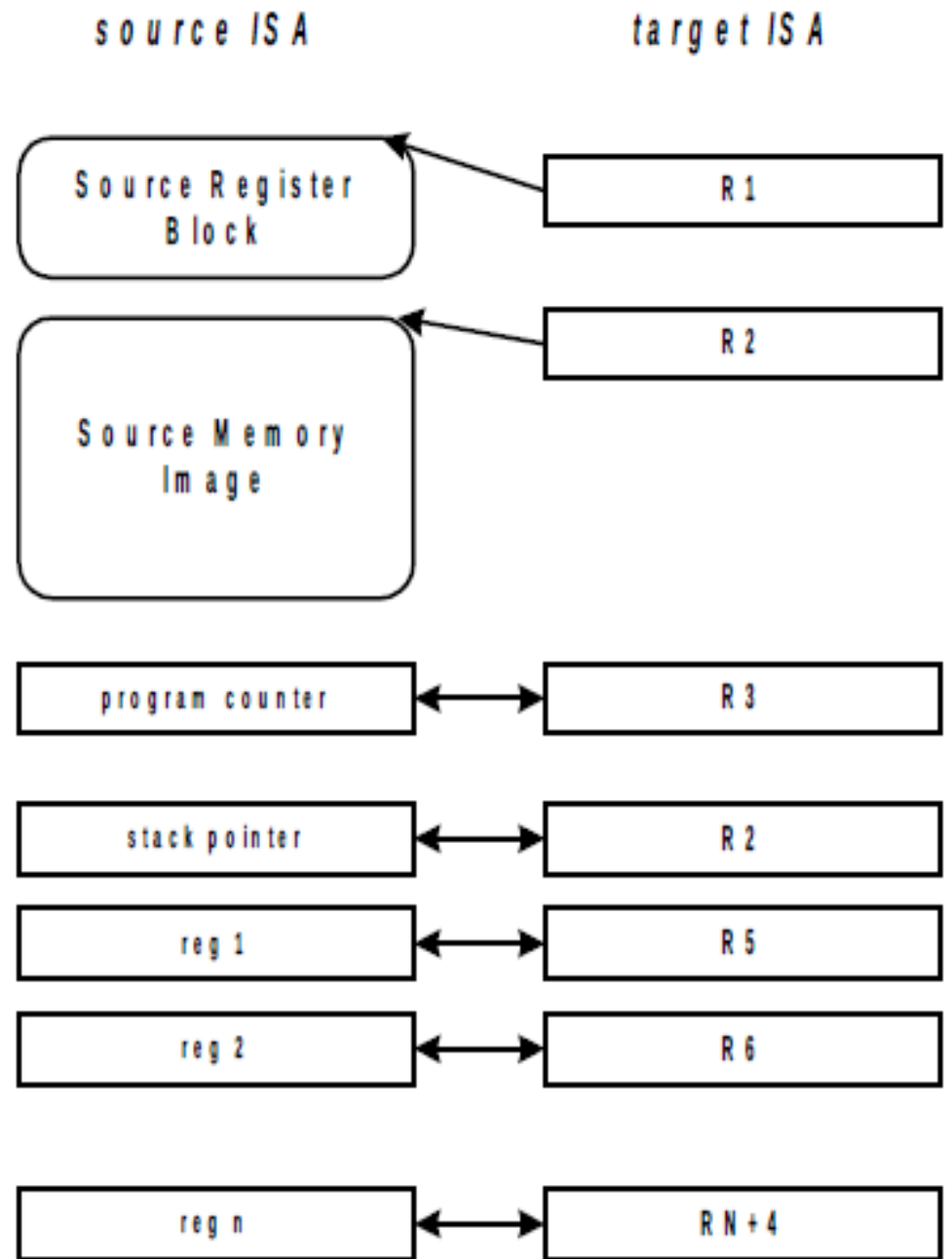


State Mapping

- Refers to the mechanism employed to maintain source program state from source ISA on the host/target ISA
- Program state
 - state includes source registers and memory contents
- Source registers can be held in host registers or in host memory
- Reduces loads/stores significantly
- Straightforward when target registers $>$ source registers

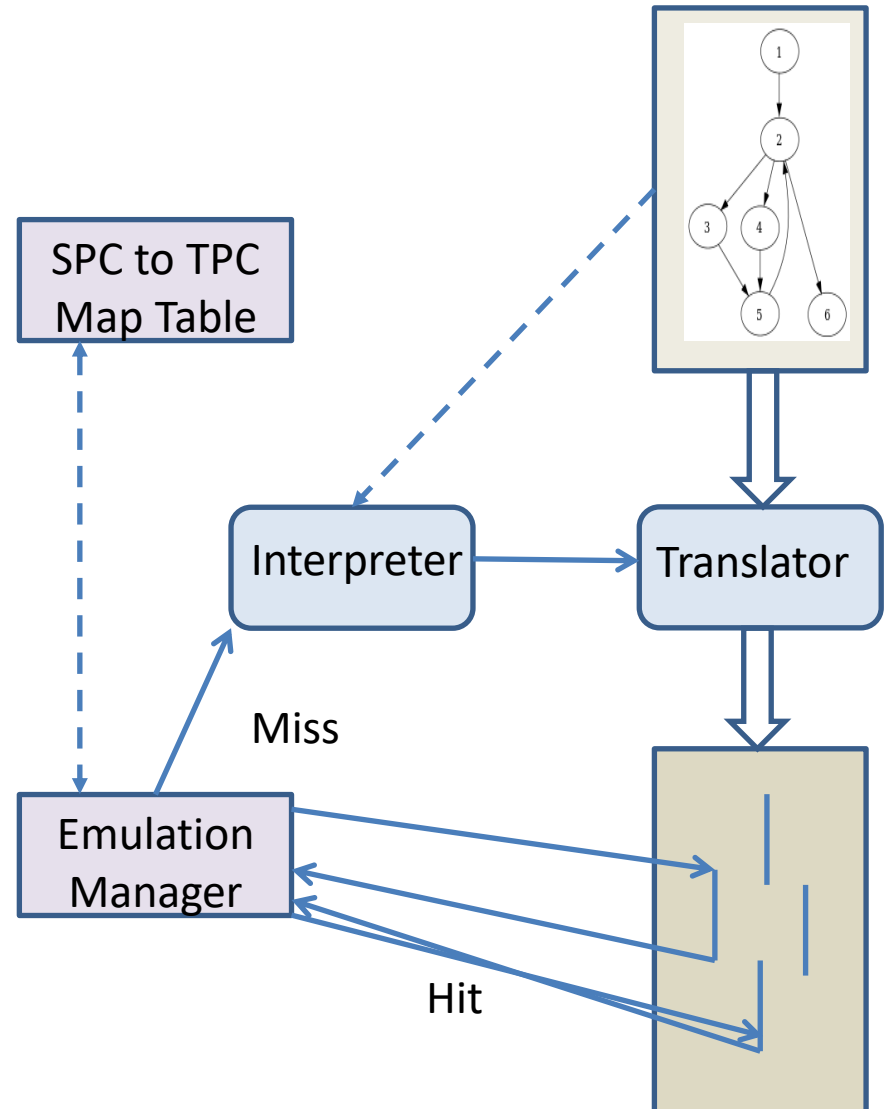
Register Mapping

- Map source registers to target registers
 - spill registers if needed
- if target registers < source registers
 - map some to memory
 - map on per-block basis



Dynamic Binary Translation

- Static binary translation makes certain assumptions about the ISA and coding practice, which may not always be true!
- Code Discovery Problem
 - Variable length instructions in case of CISC ISAs
 - Register indirect jumps
 - Data interspersed with instructions
 - Pads to align instructions
- Code-Location Problem
 - Tracking jumps in source code via the translated code and vice-versa.
- Incremental code Translation
 - First interpret
 - perform code discovery as a by-product
 - Translate code
 - incrementally, as it is discovered
 - place translated code in code cache
 - use lookup table to save source to target PC mappings
 - Emulation process
 - execute translated block
 - lookup next source PC in lookup table
 - if translated, jump to target PC
 - else, interpret and translate



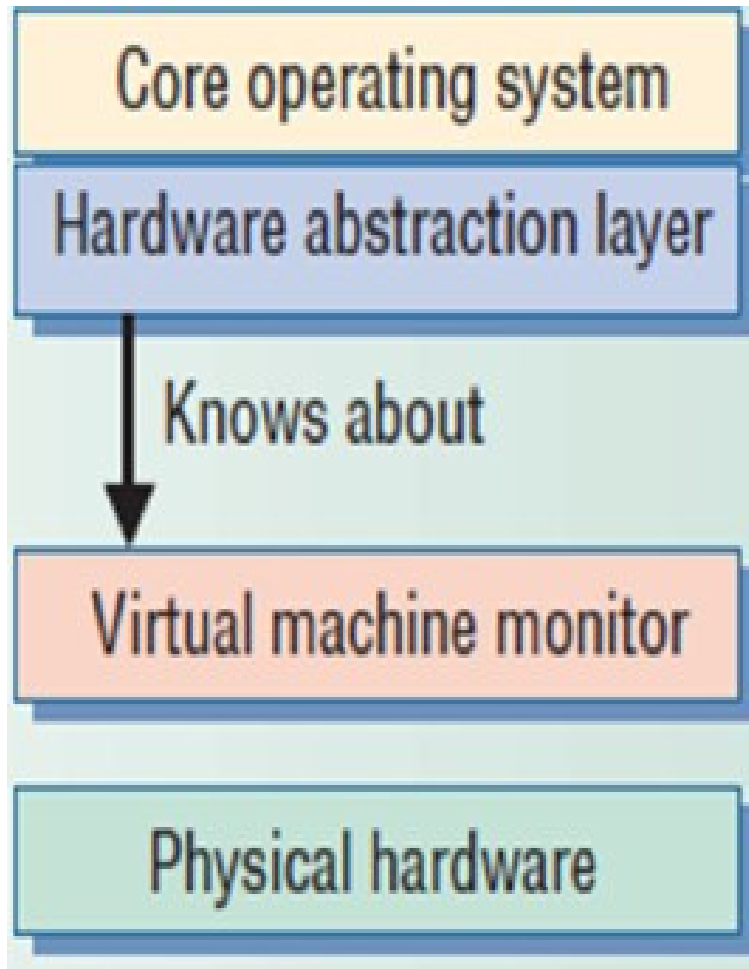
Issues with Dynamic Binary Translation

- Tracking the source PC
 - SPC used by the emulation manager and interpreter
- Handle self-modifying code
 - programs modifying code (perform stores) at runtime
- Handle self-referencing code
 - programs perform loads from the source code
- Provide precise traps
 - provide precise source state at traps and exceptions

Para-Virtualization

- OS assisted virtualization is commonly referred to as para-virtualization
 - The GuestOS is modified to exit through a hypercall (system call on a privileged instruction to the hypervisor which then completes the instruction.
 - In a non-virtualized OS device access is hidden behind the OS system call interface as these are privileged instructions.
 - OS ensures user and context isolation through the system call interface
 - Performance and security were the goals to devise para-virtualization
 - Migration and portability are issues with para-virtualization
- Examples: Xen and VmWare hypercalls and back-end with front-end device model

Para-Virtualized VMM



- To concept of a hypercall is same as that of system call in the core OS.
- Hypercalls replace the system calls in GuestOS because of de-privileging
- Hypercalls trap to VMM which then executes to complete the function of the hypercall (system call)
- GuestOS gets modified to replace the system calls with hypercalls
- Hypercall implementation need NOT follow emulation method for compatibility and implement for performance.

Hardware Virtualization

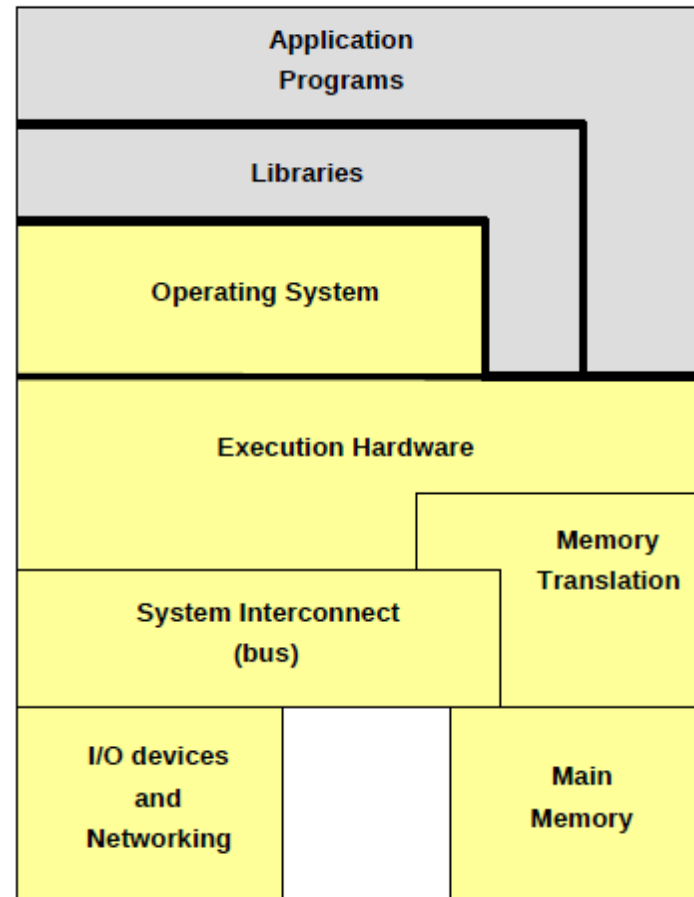
- Hardware virtualization refers to building hardware support for virtualization!
- Prevalent technology evolution has occurred mainly using software techniques for providing the virtualization support.
- Hardware constructs to support concurrent access to hardware resources.
 - CPU privilege levels
 - MMU with extended/nested page tables
 - IOV devices
- Each of these constructs allows for building a virtual device context for the resource.
- A set of these virtual devices are then exported to a VM and the GuestOS boots over the virtual devices.
- Why is there no hardware virtualizer as yet?

Goals of Process Virtualization

- Uses the concept of “process” abstraction to virtualize.
- Resource access and controls are based on the OS level resource abstractions and amenable for use without any change in hardware.
- Process abstraction:
 - Identity of access based on ownership rights
 - Virtual memory address boundaries
 - OS-system call interface for I/O device access
- Virtual Machine is executed as a process in the host OS address space.

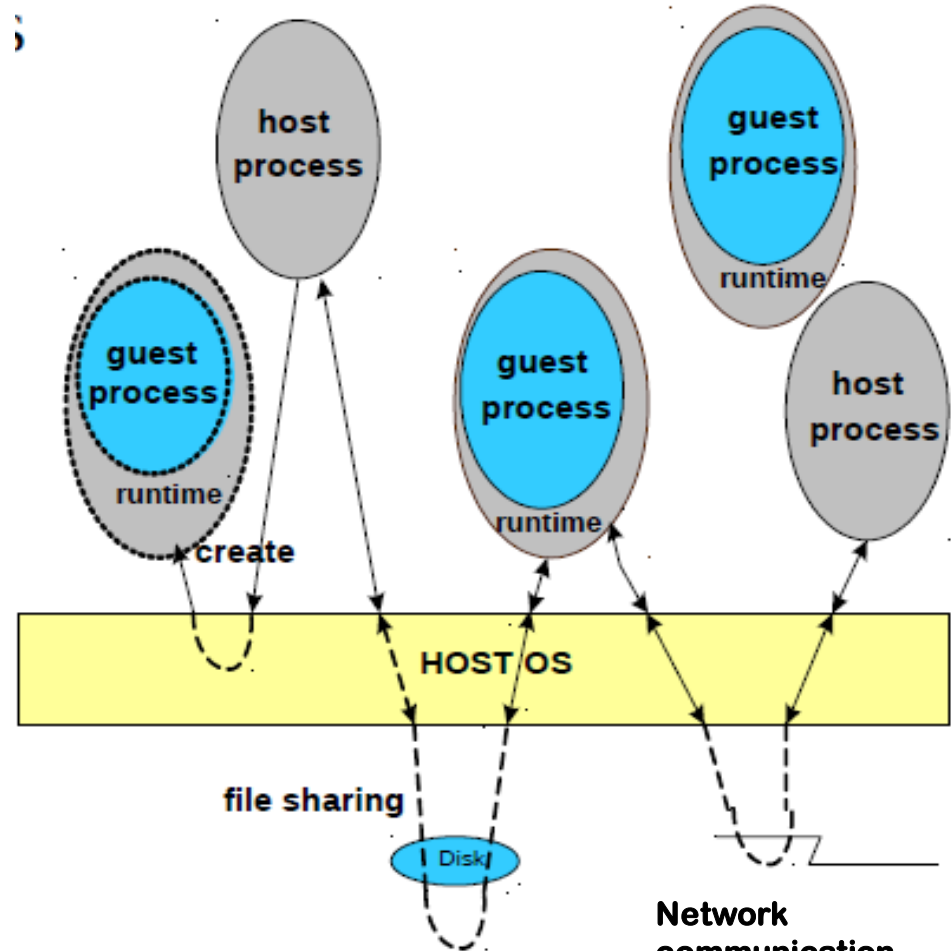
ABI Interface

- The Application Binary Interface (ABI)
 - User ISA + OS-System Calls
 - Used by compiler writers
- Platforms supporting common ABI
 - Application execution without recompilation
- Process Virtual Machine realization

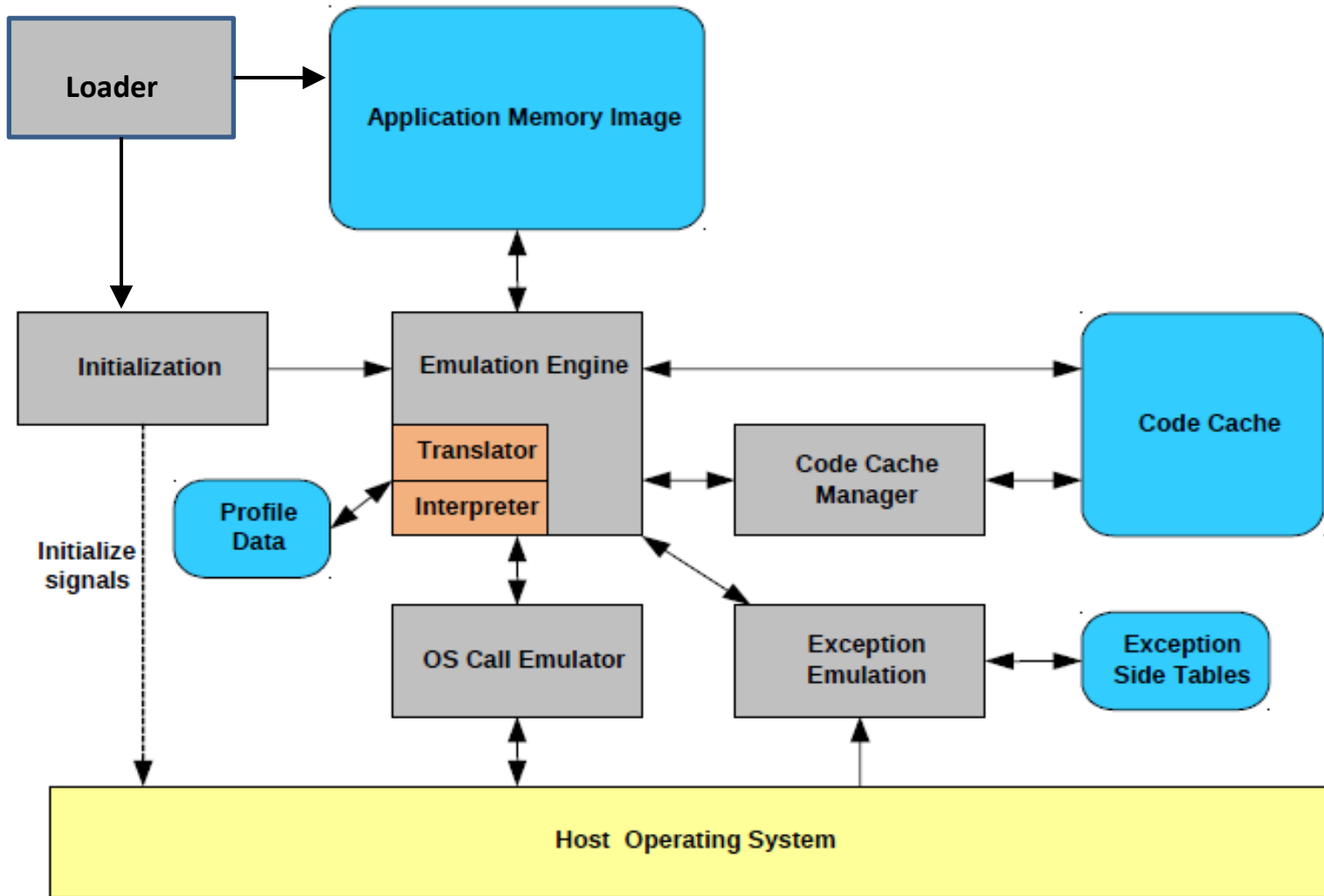


Virtualization through ABI

- Binary translation
- The Process runs as a Guest process inside a Process VM with runtime software
- The runtime software interfaces with the host process to execute guest process instructions



Process Virtual Machine Implementation

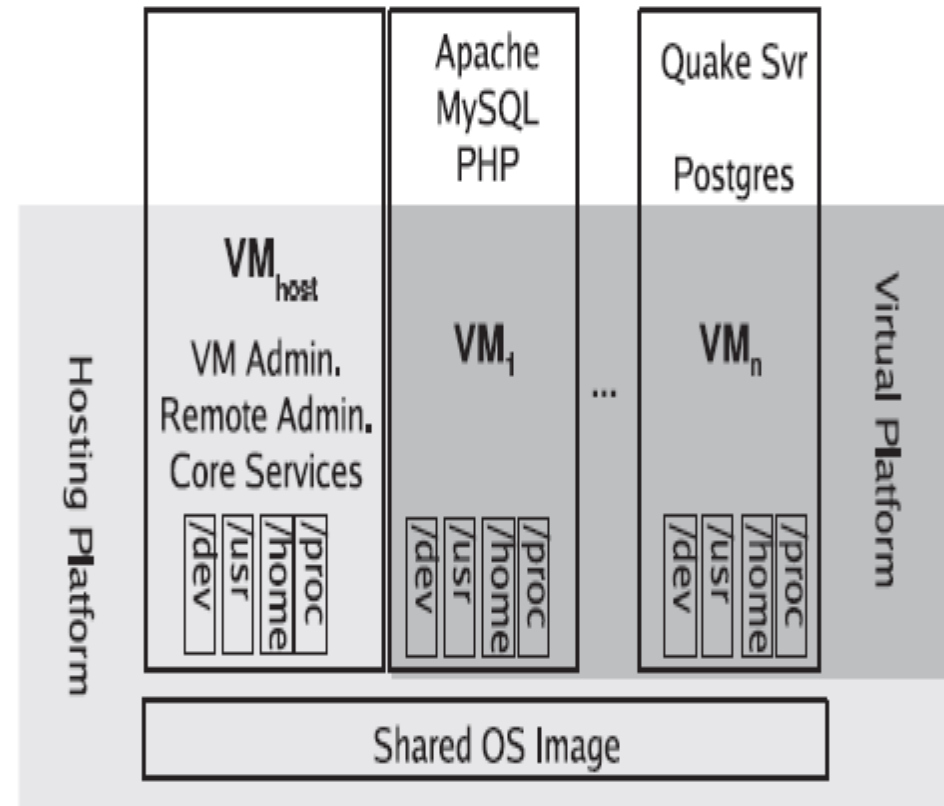


Structure of the Process VM

- Loader
 - load guest code and data
 - load runtime code
- Initialization
 - allocate memory
 - establish signal handlers
- Emulation engine
 - interpreter and/or translator
- Code cache manager
 - manage translated guest code
 - flush outdated translations
- Profile database
 - hold program profile info.
 - block/edge/invocation profile
- OS call emulator
 - translate OS calls
 - translate OS responses
- Exception emulator
 - handle signals
 - form precise state
- Side tables
 - structures used during emulation

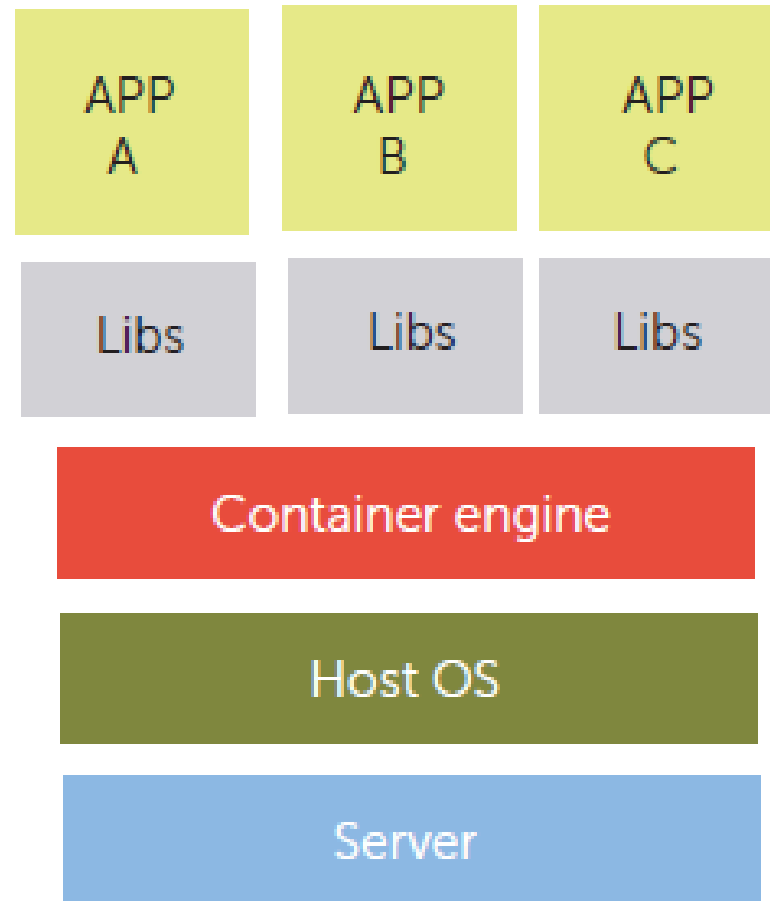
Containers as Process VMs

- Containers are logical isolated groups of processes with well defined resource controls within OS space (containment features).
- They are achieved by using extensions to OS like Solaris zones with chroot access to filesystem space or Linux container features.
 - Zones enable restricted namespace thereby limiting visibility to processes and users within the zone
 - chroot restricts access to filesystem space
- Containers are restricted Process VMs that share a common host OS and hence support same-ISA-same-OS processes or supported-ISA-similar-OS processes.



Linux Docker Containers

- Linux docker containers offer an environment close to a VM but without running a separate guest kernel and simulating the hardware.
- Linux Docker containers use a combination of kernel security features such as
 - isolated namespaces (LXCFS)
 - mandatory access control (LXC)
 - control groups (CGManager)



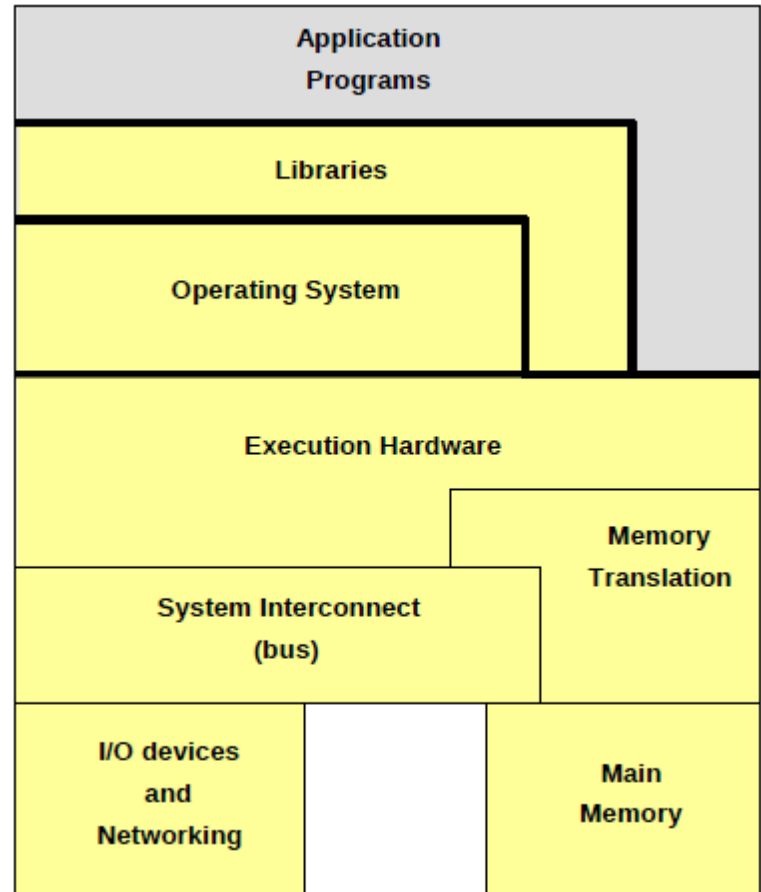
Application's Perspective of Cloud based Runtimes

| Application | | | | | | | | | | |
|-------------|----|-----------|-----------|-------------------|-------------------|------|------|------|-------------------|------|
| OS | VM | Container | Container | Virtual appliance | Virtual appliance | PaaS | PaaS | PaaS | SaaS | SaaS |
| | | | | | Virtual appliance | | | | Virtual appliance | |
| | OS | OS | OS | OS | OS | OS | OS | OS | OS | OS |
| | | | | | | | | | | |
| OS | OS | OS | OS | OS | OS | OS | OS | OS | OS | |

Figure 2. Possible layering combinations for application runtimes.

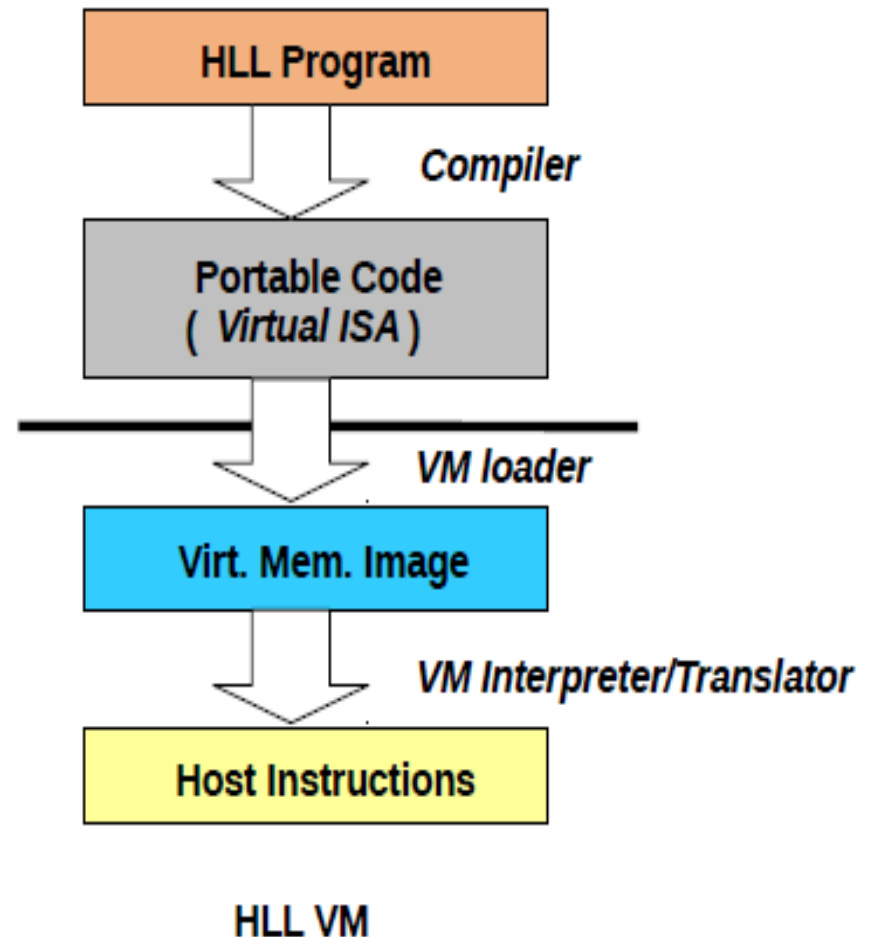
API Interface

- Application Programming Interface (API)
 - User ISA + Library calls
 - Used by application programmers
- Realization of HLL based virtual machines



Motivation for API based VMs

- Application portability is the key goal behind using API layer for virtualization.
- Conventional ISAs not built for virtualization!
 - Most constructs are after thoughts for supporting virtualization.
 - Portability and performance are seemingly orthogonal goals to achieve!
- Virtual ISA
 - Primary design goal is VM based portability
 - V-ISA is not designed for real hardware processor!



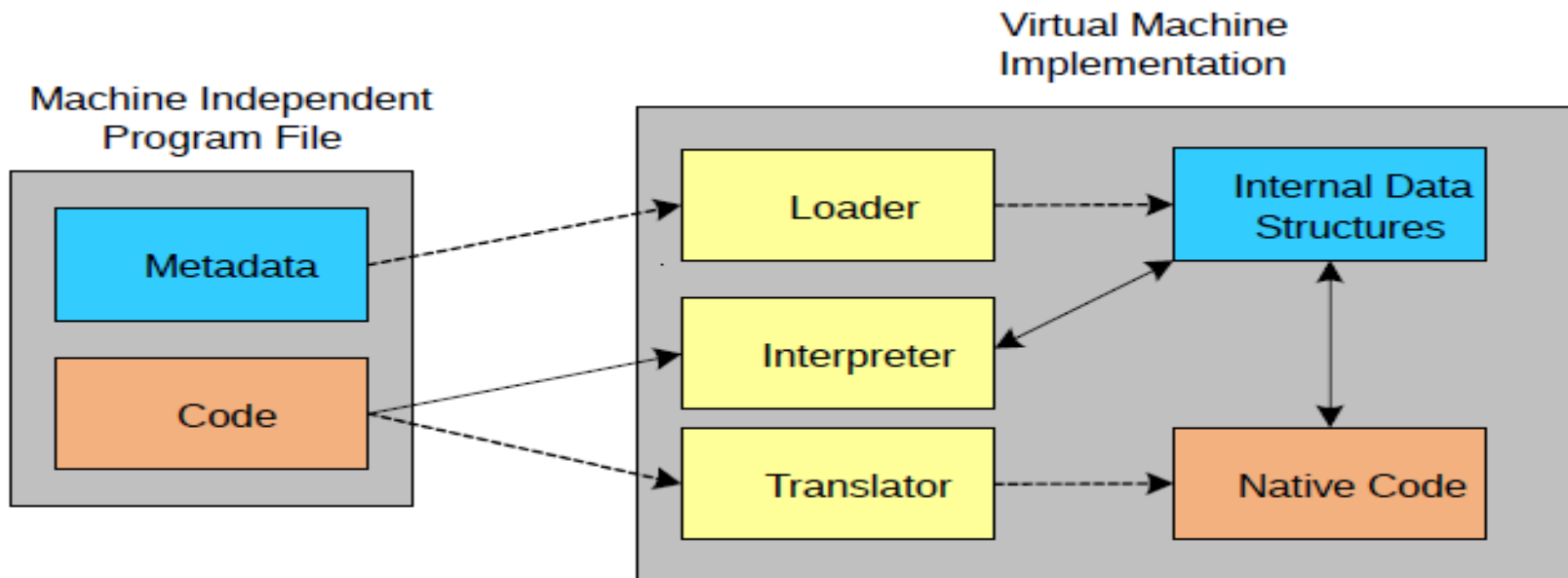
Virtualization using APIs (HLL VMs)

Examples

- Pascal P-code
- JVM
- CLI/.Net framework

Virtualization using APIs (HLL VMs)

- Compiler frontend produces binary files that is common to all ISAs
- Binary files are intermediate code that is interpreted or translated at runtime.
- Contains both code and metadata



Genesis of V-ISAs

- Virtualizing using conventional ISAs:
 - Not built for virtualization
 - OS dependencies
 - Address-space and page size limitations
 - ISA features and memory formation coupled
 - Maintaining precise exceptions
 - Instruction discovery during indirect jumps
 - Self-referencing and self-modifying code poses problems for translated code
- HLL V-ISA
 - Design goal is virtualization
 - Generous use of metadata
 - Metadata allows for type safe code verification, interoperability and performance
 - Reduced OS dependencies as programs interact through library API
 - Abstract memory model of indefinite size; memory regions allocated based on needs; actual memory addresses not visible

V-ISAs compared to C-ISAs

- Conventional ISA
 - unrestricted address computation
 - difficult to protect runtime from unauthorized guest program accesses
 - Trap and exceptions need precise state; difficult to emulate
 - guest ISA registers > host registers is a problem
 - ISAs with condition codes are difficult to emulate
 - indirect jumps to potentially arbitrary locations
 - variable-length instruction, embedded data, padding
- HLL V-ISA
 - pointer arithmetic not permitted
 - memory access only through explicit memory pointers
 - static/dynamic type checking employed
 - Traps are limited and exceptions are tested within the programs; precise exception state requirement is relaxed
 - stack-oriented; condition codes are avoided
 - restricted indirect jumps
 - no mixing of code and data; variable-length instructions permitted
 - self-modifying and self-referencing code not permitted

Summary

- Goals for Virtualization
- Basic Requirements and constructs used for
 - System VMs
 - Process VMs
- Shortfalls and advantages of the constructs