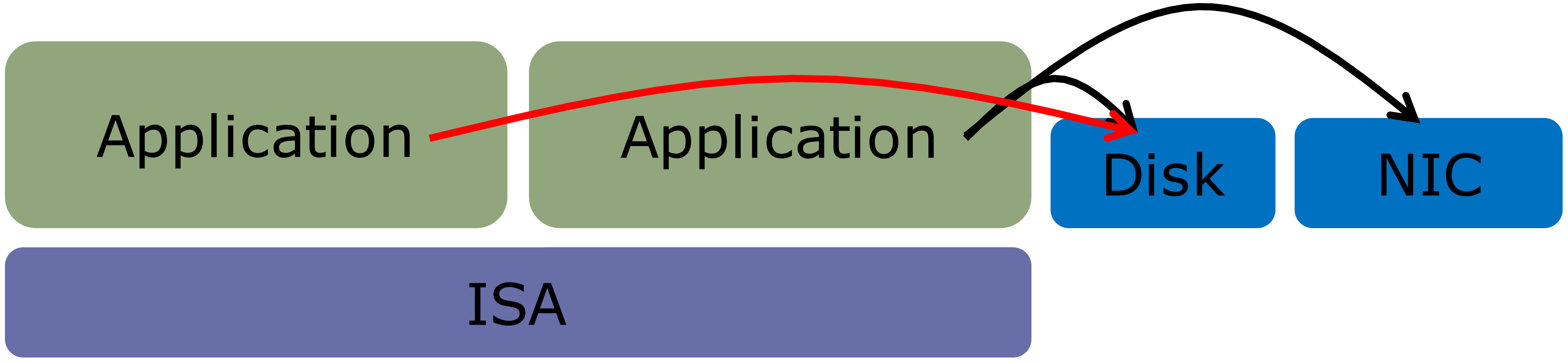
Virtual Machines and Binary Translation

Nathan Beckmann
(based on slides by David Wentzlaff and Nathan Binkert)

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

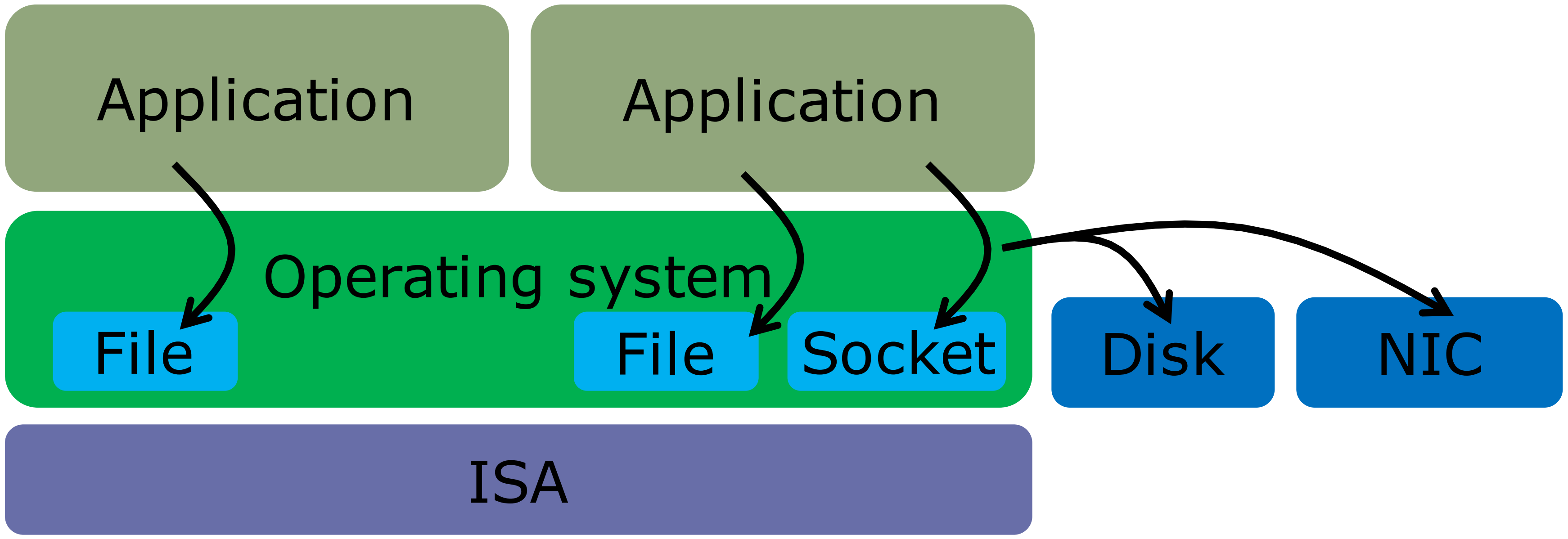
What is a “machine”?

- Is ISA sufficient?
- No! Need access to outside world (I/O)
- But applications contend for devices



Operating system handles I/O

- Operating system abstracts—or *virtualizes*—hardware I/O
  - E.g., files for disk, sockets for network
  - Hides differences between devices (e.g., tape vs disk vs SSD)
  - Provides protection & security



ISA+Environment = Virtual machine

- ISA defines available operations
  - But insufficient alone to write useful programs
- Operating system (OS) responsible for I/O
  - Protection, portability, and abstraction

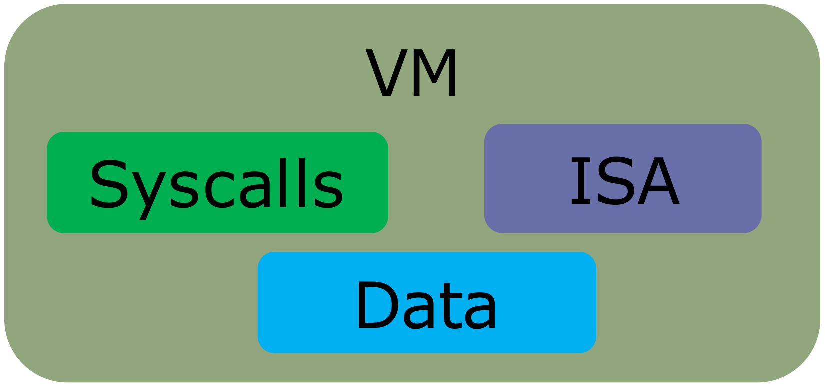
- ISA communicates with operating system through some standard mechanism, i.e., `syscall` instructions
  - Example: opening a file

```
addi r1, r0, 27             # 27 is code for file open
addu r2, r0, rfname         # r2 points to filename string
syscall                     # cause trap into OS
# On return from syscall, r1 holds file descriptor
```

Application binary interface (ABI)

- Programs distributed in a binary format:
  - Instructions
  - Initial values (in *data segments*)

- Virtual machine specified by:
  - Which instructions are available (ISA)
  - What system calls are possible (I/O, or the *environment*)
  - What state exists at start of day



- Operating system implements the VM
  - (1) loads binary
  - (2) creates environment
  - (3) begins execution
  - (4) handles traps for syscalls, unsupported instructions, etc.

OS can support multiple VMs

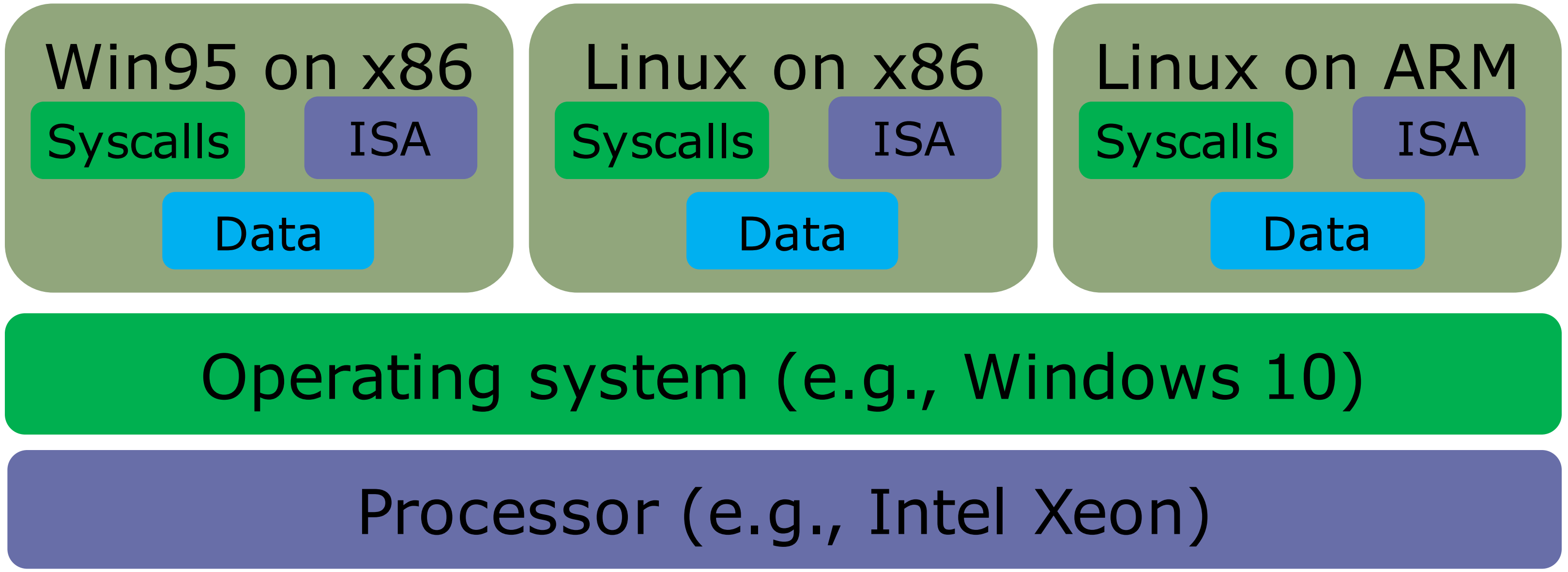
- Virtual machine features change over time
  - New instructions
  - New types of I/O (e.g., asynchronous file I/O)

- Common to provide backwards compatibility
  - Linux v4.5 (2016) can run binaries compiled for Linux v1.0 (1994)
  - Windows runs MS-DOS binaries
  - Windows 10 runs Linux binaries
  - Linux runs Windows programs (through Wine)

- If ABI needs instructions not supported by native hardware, then *OS can emulate them in software*

OS can support multiple VMs

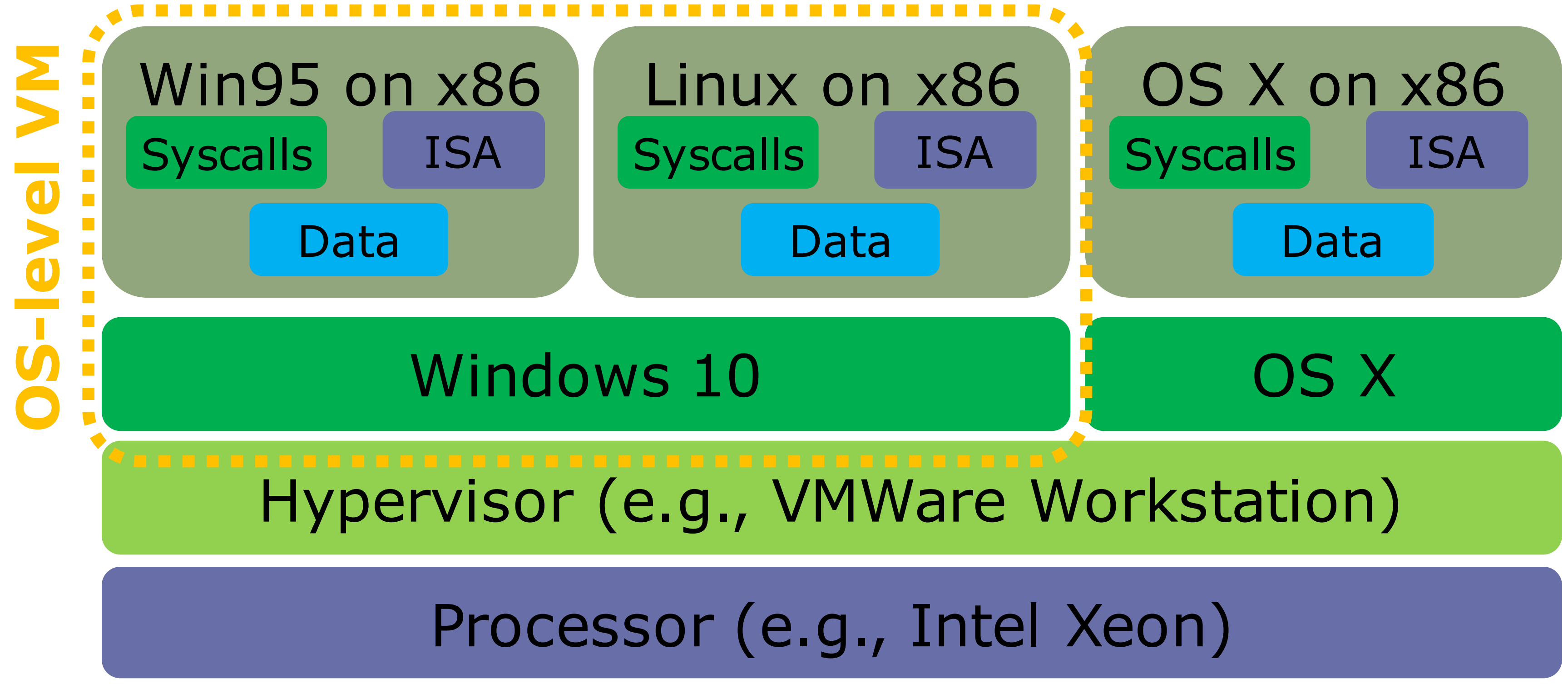
- Virtual machine features change over time
- Common to provide backwards compatibility
- If ABI needs instructions not supported by native hardware, then *OS can emulate them in software*



Hardware can support multiple OSes

- Can virtualize the environment that an operating system sees, i.e. an *OS-level* VM
  - Probably what you are familiar with!
- *Hypervisor* abstracts real hardware resources, letting each OS VM think it has the full machine
  - Popular in early days to allow mainframe to be shared by multiple groups developing OS code (VM/370)
  - Used in modern mainframes to allow multiple versions of OS to be running simultaneously → OS upgrades with no downtime!
  - Example for PCs: VMware allows Windows OS to run on top of Linux (or vice-versa)
- Requires trap on access to privileged hardware state
  - Easier if OS-hardware interface is well defined

Hardware can support multiple OSes



Nomenclature

From (Machine or ISA we are attempting to execute)

- Guest
- Client
- Foreign ISA

To (Machine that is doing the real execution)

- Host
- Target
- Native ISA

ISA Implementations Partly in Software

Often good idea to implement part of ISA in software:

- Expensive but rarely used instructions can cause trap to OS emulation routine:
  - e.g., decimal arithmetic in μVax implementation of VAX ISA
- Infrequent but difficult operand values can cause trap
  - e.g., IEEE floating-point denormals cause traps in almost all floating-point unit implementations
- Old machine can trap unused opcodes, allows binaries for *new* ISA to run on *old* hardware
  - e.g., Sun SPARC v8 added integer multiply instructions, older v7 CPUs trap and emulate

Supporting Non-Native ISAs

Run programs for one ISA on hardware with different ISA

Binary ***emulation***:

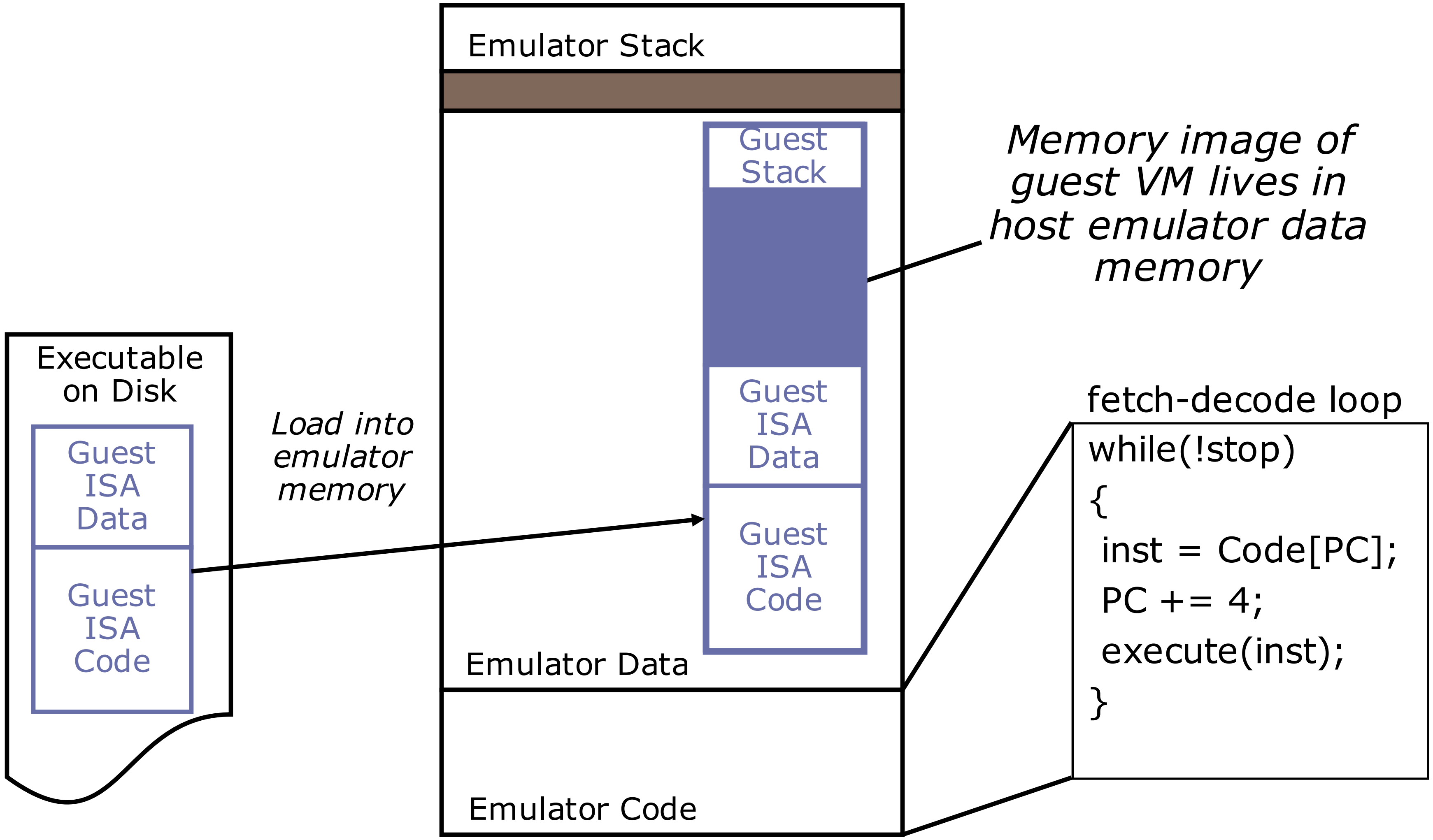
- Hardware
  - IBM System 360 had IBM 1401 emulator in microcode
  - Intel Itanium converts x86 to native VLIW (two software-visible ISAs)
  - ARM cores support 32-bit ARM, 16-bit Thumb, and JVM (three software-visible ISAs!)
- Software *(OS interprets instructions at run-time)*
  - PowerPC Macs had emulator for 68000 code

Binary ***translation***:

- Static *(convert at install time, load time, or offline)*
  - IBM AS/400 to modified PowerPC cores
  - DEC tools for VAX->Alpha and MIPS->Alpha
  - Android
- Dynamic *(non-native ISA to native ISA at run time)*
  - Sun's HotSpot Java JIT (just-in-time) compiler
  - Transmeta Crusoe, x86->VLIW code morphing
  - PIN!

Software emulation

- Software instruction set interpreter fetches and decodes one instruction at a time in emulated VM



```
while(!stop)
{
 inst = Code[PC];
 PC += 4;
 execute(inst);
}
```

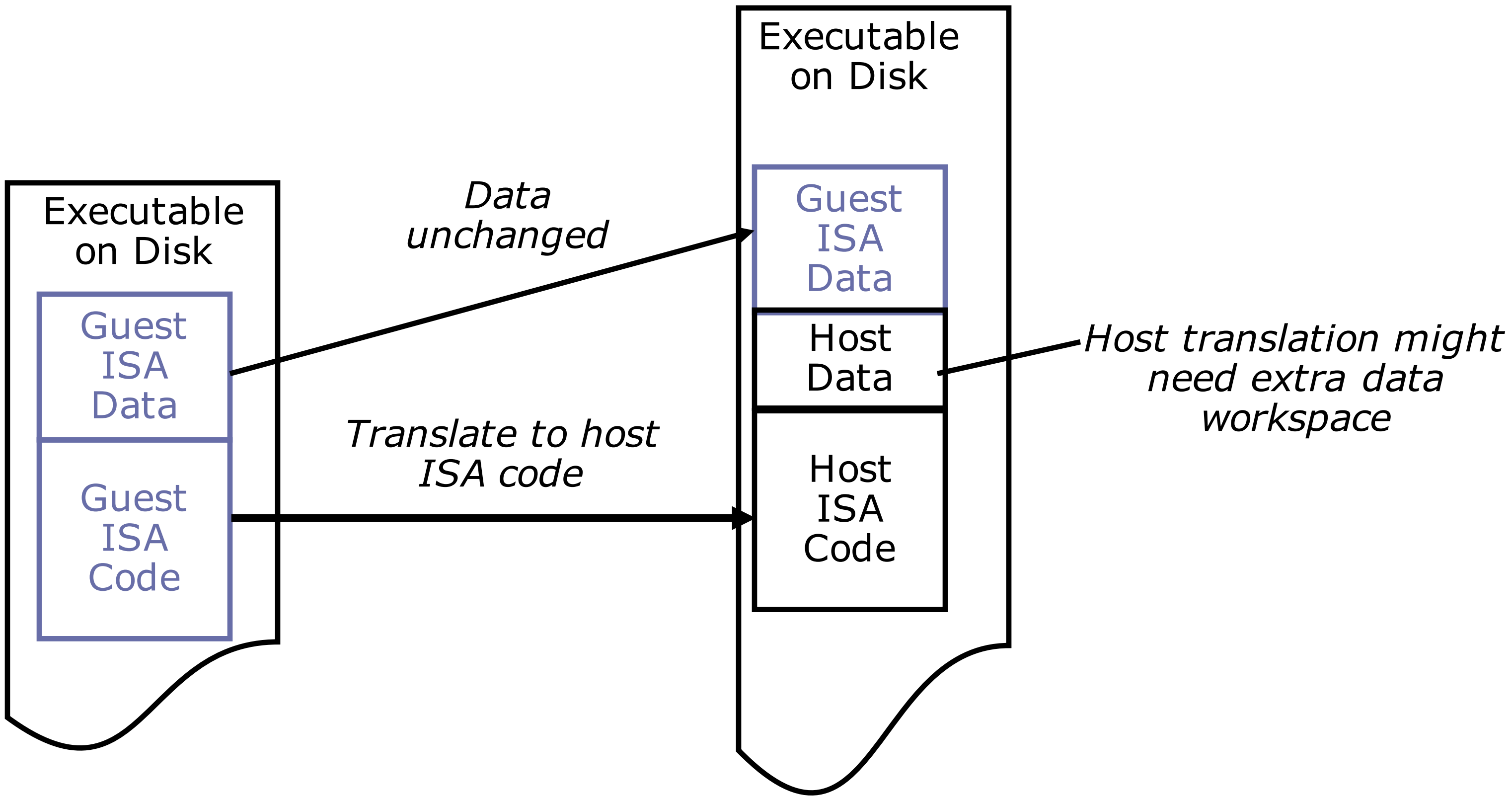
Software emulation

- Easy to code, small code footprint
- ***Slow***, approximately 100x slower than native execution for RISC ISA hosted on RISC ISA
- Software decode is the problem—must:
  - fetch instruction from memory
  - switch tables to decode opcodes
  - extract register specifiers using bit shifts
  - access register file data structure
  - *then* execute operation

Translation

- Each guest ISA instruction translates into some set of host (or *native*) ISA instructions
- Instead of dynamically fetching and decoding instructions at run-time, ***translate*** entire binary program and save result as new host ISA executable
- Removes interpretive fetch-decode overhead
- Can re-optimize translated code
  - register allocation
  - instruction scheduling
  - inline assembly
  - remove dead or unreachable code

Static Binary Translation (SBT), Take 1



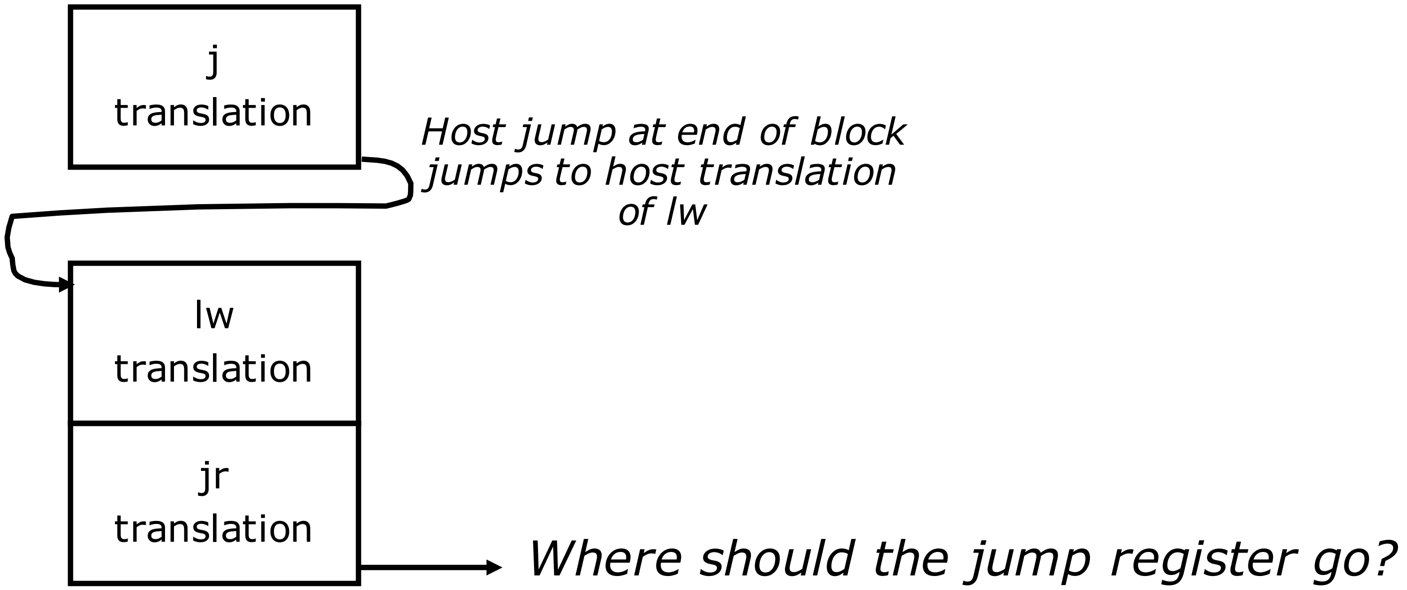
SBT Problem: Indirect jumps

Branch and Jump targets

- guest code:

```
        j L1
          ...
    L1: lw r1, (r4)
        jr (r1)
```

- host code



Where should the jump register go?

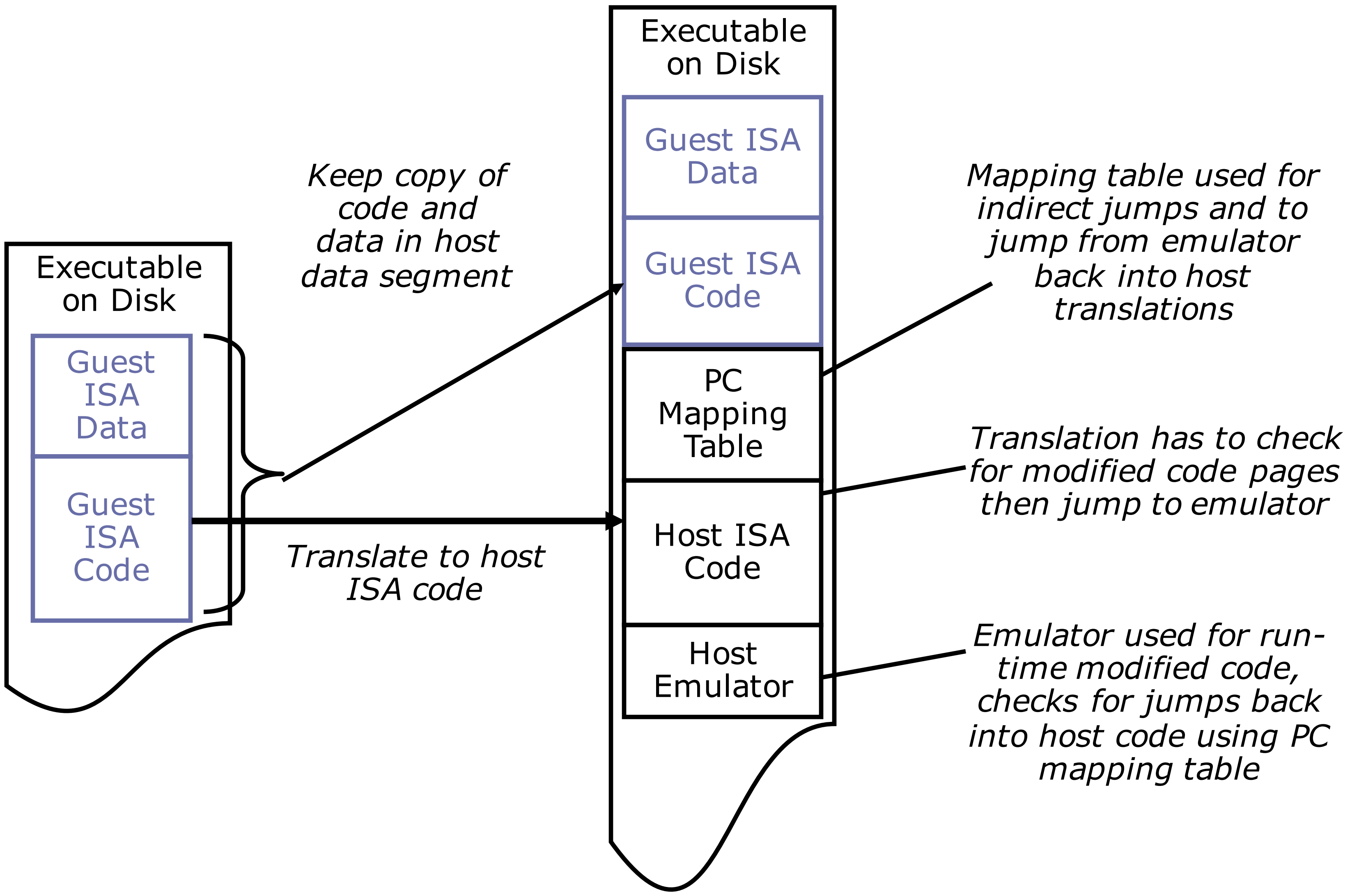
SBT Solution: PC Mapping Table

- Table gives translated PC for each guest PC
- Indirect jumps translated into code that looks in table to find where to jump to
  - can optimize well-behaved guest code for subroutine call/return by using host PC in return links
- If can branch to any guest PC, then need one table entry for every instruction in hosted program ➔ *big table*
- If can branch to any PC, then either
  - limit inter-instruction optimizations
  - large code explosion to hold optimizations for each possible entry into sequential code sequence
- But only minority of guest instructions are indirect jump targets, want to find these
  - Structure ISA to disallow problems (e.g., only subroutine calls)
  - Dynamic, feedback-guided translation?

More SBT problems: Self-modifying code

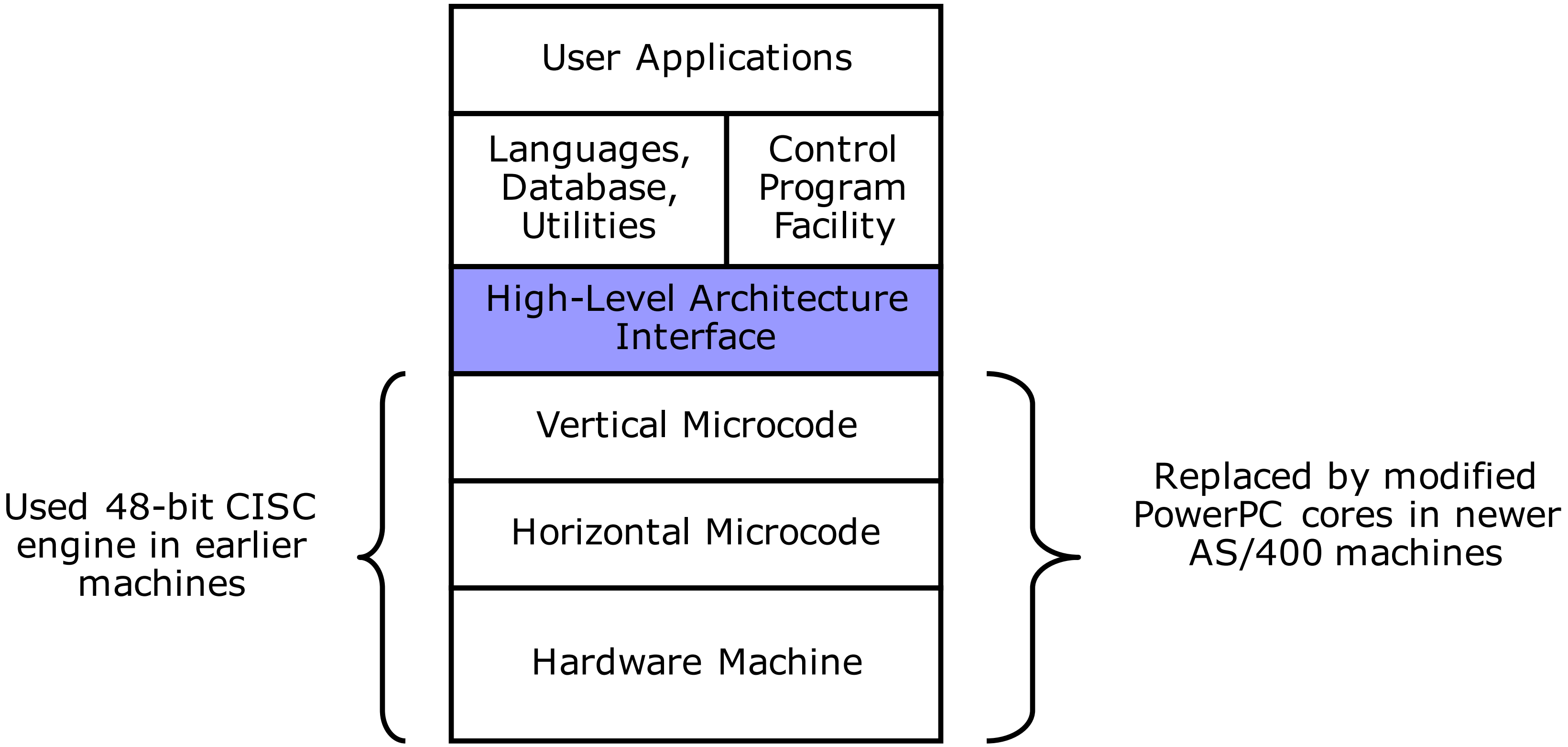
- Rare in most code, but has to be handled if allowed by guest ISA
- Usually handled by including interpreter and marking modified code pages as “interpret only”
- Have to invalidate all host branches into modified code pages

Static Binary Translation, Take 2


Executable on Disk
Guest ISA Data
Guest ISA Code
Keep copy of code and data in host data segment
Translate to host ISA code
Executable on Disk
Guest ISA Data
Guest ISA Code
PC Mapping Table
Host ISA Code
Host Emulator
Mapping table used for indirect jumps and to jump from emulator back into host translations
Translation has to check for modified code pages then jump to emulator
Emulator used for run-time modified code, checks for jumps back into host code using PC mapping table

Static Translation Example: IBM System/38 and AS/400

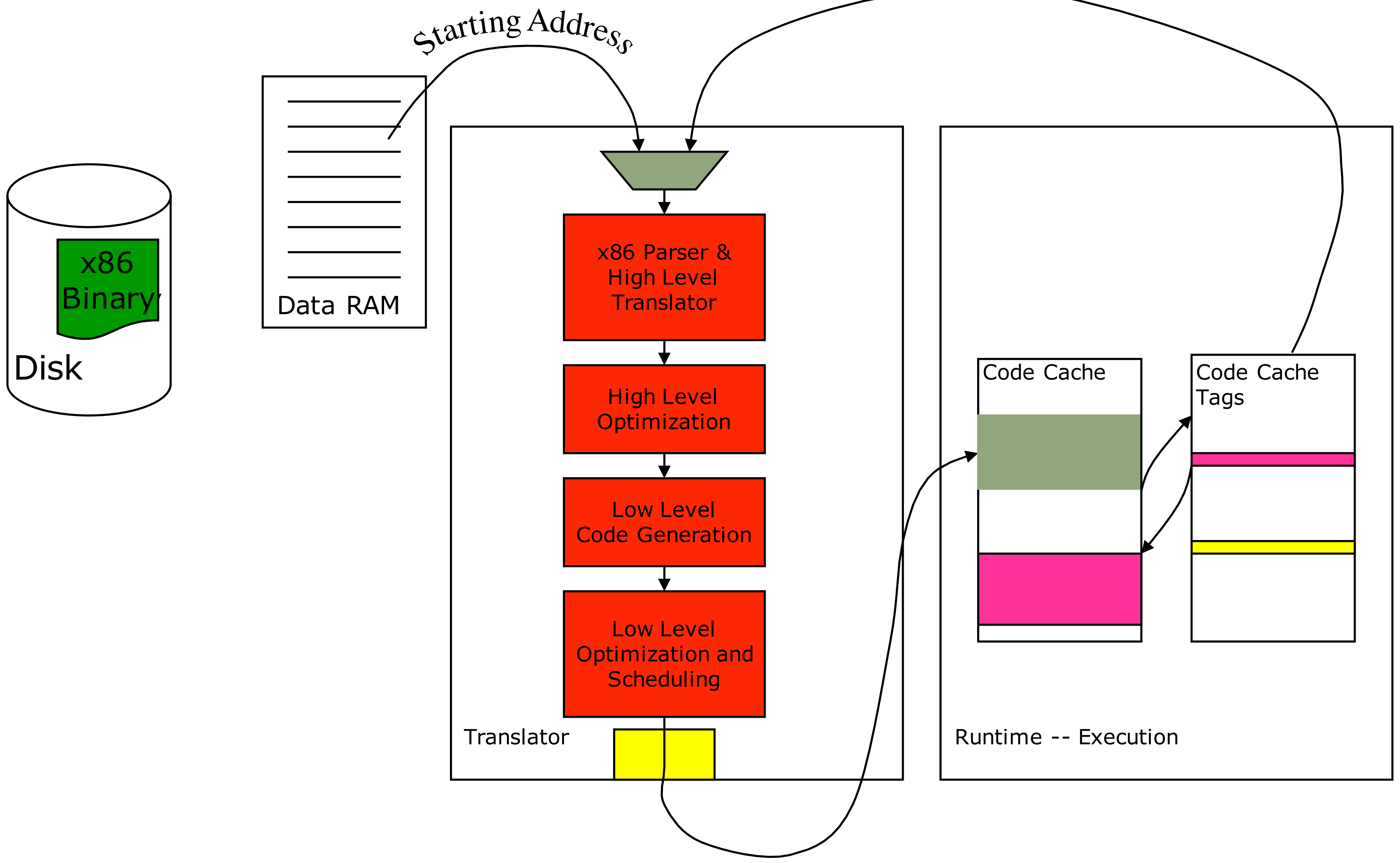
- System/38 announced 1978, AS/400 is follow-on line
- High-level instruction set interface designed for binary translation
- Memory-memory style instruction set, *never directly executed by hardware*
- Translated executable stored on disk


User Applications
Languages, Database, Utilities
Control Program Facility
High-Level Architecture Interface
Vertical Microcode
Horizontal Microcode
Hardware Machine
Used 48-bit CISC engine in earlier machines
Replaced by modified PowerPC cores in newer AS/400 machines

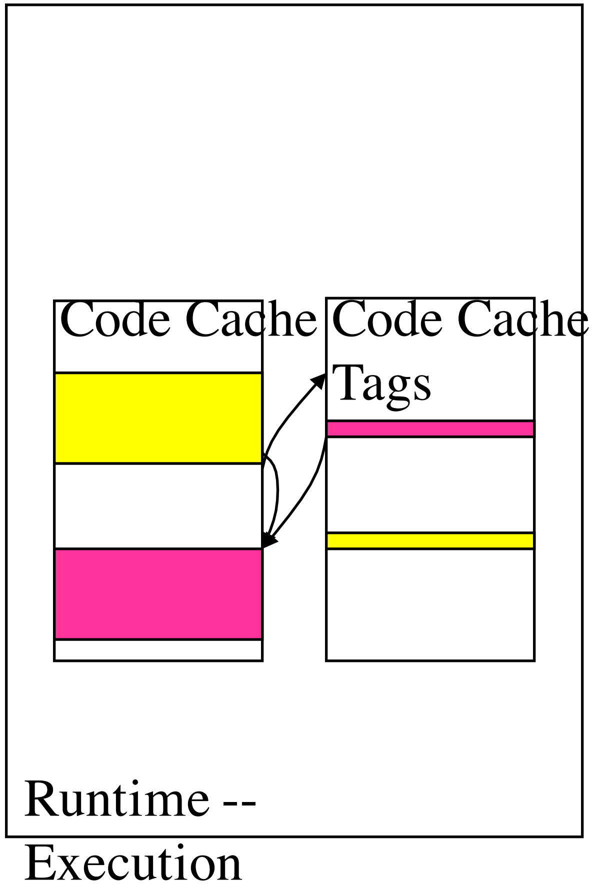
Dynamic Binary Translation

- Translate code sequences ***on demand at run-time***, caching the translations
- Can optimize code sequences based on dynamic information (e.g., branch targets encountered)
- Tradeoff between optimizer run-time and time saved by optimizations in translated code
- Used in JIT (just-in-time) compilers, PIN, Transmeta Crusoe for x86 emulation

Dynamic Translation Example



Making DBT efficient: Chaining



Pre Chained

```
add %r5, %r6, %r7
li %next_addr_reg, next_addr #load address
                                  #of next block
j dispatch loop
```

Chained

```
add %r5, %r6, %r7
j physical location of translated
        code for next_block
```

Dynamic Translation Example: Transmeta Crusoe (2000)

- Converts x86 ISA into internal native VLIW format using software at run-time ➔ “Code Morphing”
- Optimizes across x86 instruction boundaries to improve performance
- Translations cached to avoid translator overhead on repeated execution
- Completely invisible to operating system – looks like x86 hardware processor

Full virtualization vs. Paravirtualization

- Full virtualization: guest OS is totally unaware of virtualization
  - E.g., IBM 360
  - Full binary-level compatibility
  - Implemented in either hardware or software
  - Can hurt performance

- Paravirtualization: guest OS is aware & cooperates
  - E.g., Xen
  - Special drivers to "play nice" with other guests, coordinated thread & memory management
  - Necessary for ISAs/devices that are hard to virtualize
  - Higher performance, but requires extensive OS modifications

When is an architecture virtualizable? *[Popek and Goldberg, '74]*

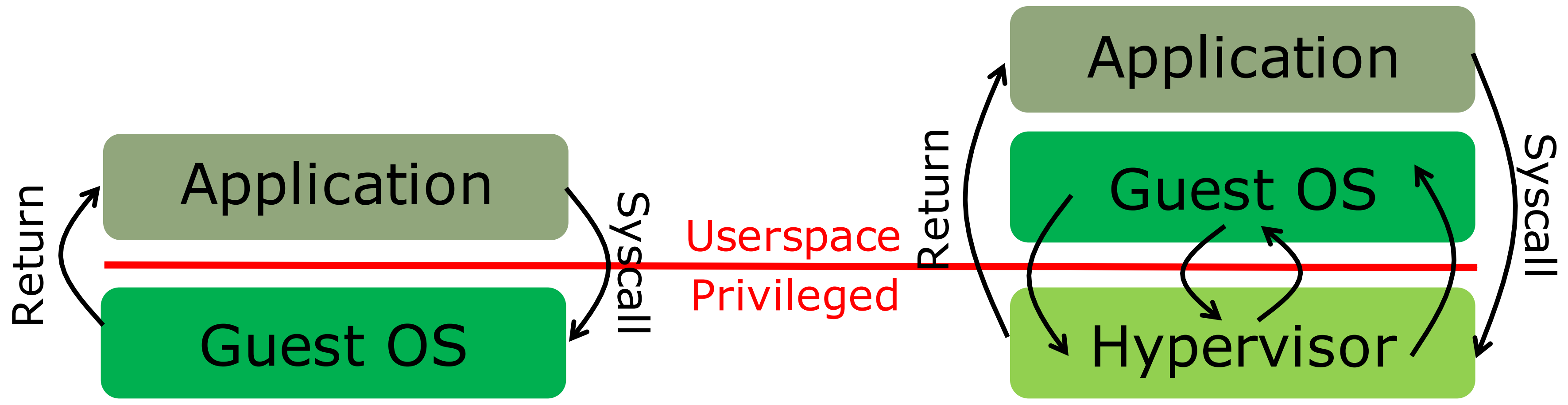
Two basic requirements:

- Privileged execution mode
- All sensitive instructions must be privileged
  - Those that change or depend on environment

To virtualize:

- Run guest OS in userspace
- Privileged instructions will trap
- Then, *just emulate them in the kernel*
  - Memory accesses must go through privileged translation (e.g., paging), architectures can provide more support for performance

Full virtualization in software



Guess OS runs in user space
Hypervisor checks permissions

Privileged instructions trap into hypervisor

What ISAs are virtualizable?

- IBM Power: yes
  - Three modes: user, supervisor, hypervisor
  - Power5, Power6 always run hypervisor
- Sun Sparc: yes
  - Similar to IBM
  - Sparc v9 always run hypervisor
- X86: Not quite
  - ~17 sensitive but unprivileged instructions
  - What to do? ➔ Binary translation! Paravirtualization!

Hardware virtualization

- Additional privilege mode for hypervisor
  - New instructions to enter/exit hypervisor
  - Swap registers + address space atomically
- Nested page tables
  - Convert guest virtual ➔ guest physical ➔ host physical
  - One TLB miss can take 24 memory accesses in x86-64!
  - Hardware page table walks
- IOMMU
  - Software emulation of I/O expensive – many copies
  - Support DMA from device to VM’s address space

That’s all, folks!