CS294
Virtual Machines
and Managed Runtimes

Mario Wolczko
Architect, Oracle Labs

TA: Patrick Li

Jonathan Bachrach

© 2015 Oracle, Mario Wolczko

The following is provided for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

Course Goal

To teach you how Virtual Machines are built today, and to guide you through building one of your own.

How we'll get there

- I'll walk you through the literature and history of how we got here.
- You'll implement examples of key ideas

  But overall, details are overwhelming

- The examples will fit together, resulting in a (fast) VM for a toy language.
  - We'll fiddle with the language and the implementation to illustrate problems from real languages

Weekly agenda

- ~2 hours of presentation; some guest lectures maybe some Q&A
- Review lab assignments and results
- Review literature — reading assignments
- Occasional student presentations

Assessment

First ground., not experienced, maybe have dynamic ddl.

- 75% of the assessed grade will come from results of lab exercises.
- The remaining 25% will come from reports and presentations and other in-class activity.

Course Non-goals

- Exhaustive coverage — this field is too big to cover everything done in it. Even subfields would merit a course of their own (e.g., garbage collection)

  a really hot pot. many can have 124 courses

- This is ***not*** a compiler course
- Objectivity — this is what **I** know and believe. I don't get to study/play with competitors' systems in the same depth as my own.
  - Example: I cover language VMs in much more depth than system VMs

FAQ #1.
Who the heck are you?

instructor

- An Architect (senior technical grade) at Oracle, formerly Sun
- Have worked on VMs for 30 years, on and off:
  - Smalltalk VM for Master's thesis, 1983–4 (Manchester)
  - Smalltalk research machine, "mushroom" 1987–93
  - Self VM @ Sun, '93–5
  - various Java VMs, '96–01
  - Also involved with: Maxine research VM, binary translation research VM
  - Founded VM Research Group, Oracle Labs

FAQ #2: Why are you doing this?

Instructor

Hard to hire people

- More people need to know this stuff!
- VMs underpin much of today's computing
  - About half of all commercial software development is done in a managed language
  - And yet relatively few people know how they work, and fewer yet how to build them
- **It's fun**

FAQ #3: Why is Oracle paying you to do this?

- We need more people to know this stuff!
- Oracle isn't just a database company. VMs are a core technology. We employ many people to work on VMs in production and research, and the candidate pool continues to be desperately small.

FAQ #4: What am student I going to get out of this?

- You will join an elite club of people knowing about VM internals (shocking, given how widely VMs are used)
- …An intern job offer?
- Ideas for interesting (and valuable!) research topics
  - …And then a regular job offer?
- You will have **fun** (I hope)

Who are you?

- Your name Hale
- 3 most proficient programming languages

  Java, Scala, python, cpp, Rust
  - Your favorite language?

    Rust
- The program (whole or part) of which you are most proud

  Crigrid compiler.

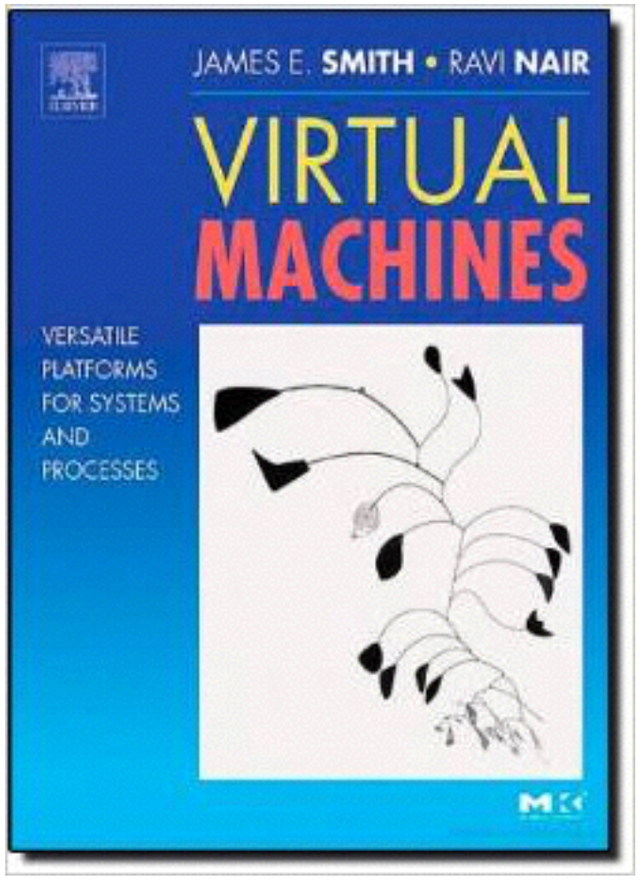
1. Introduction to VMs

Some introductory terminology

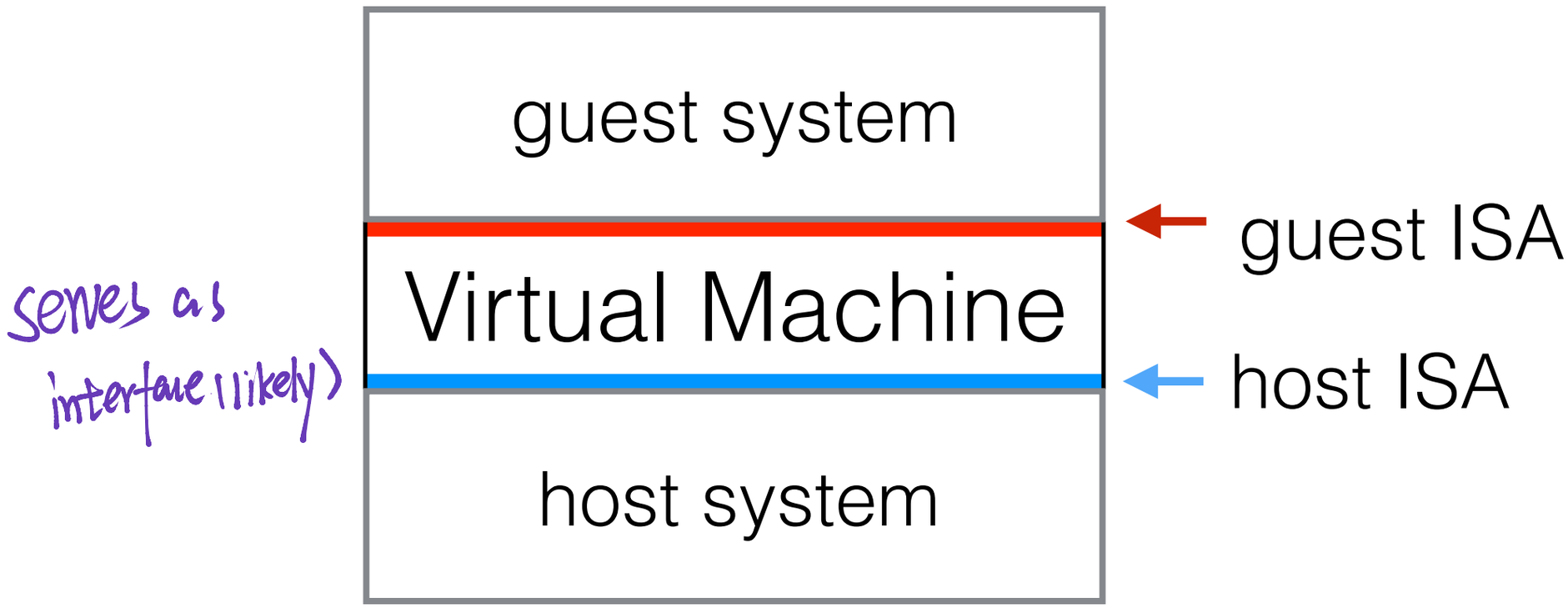
Background reading

- Smith and Nair, *Virtual Machines*, 2005
  I borrow some of their terminology and notation from Chapter 1

- Will introduce papers and other books when appropriate



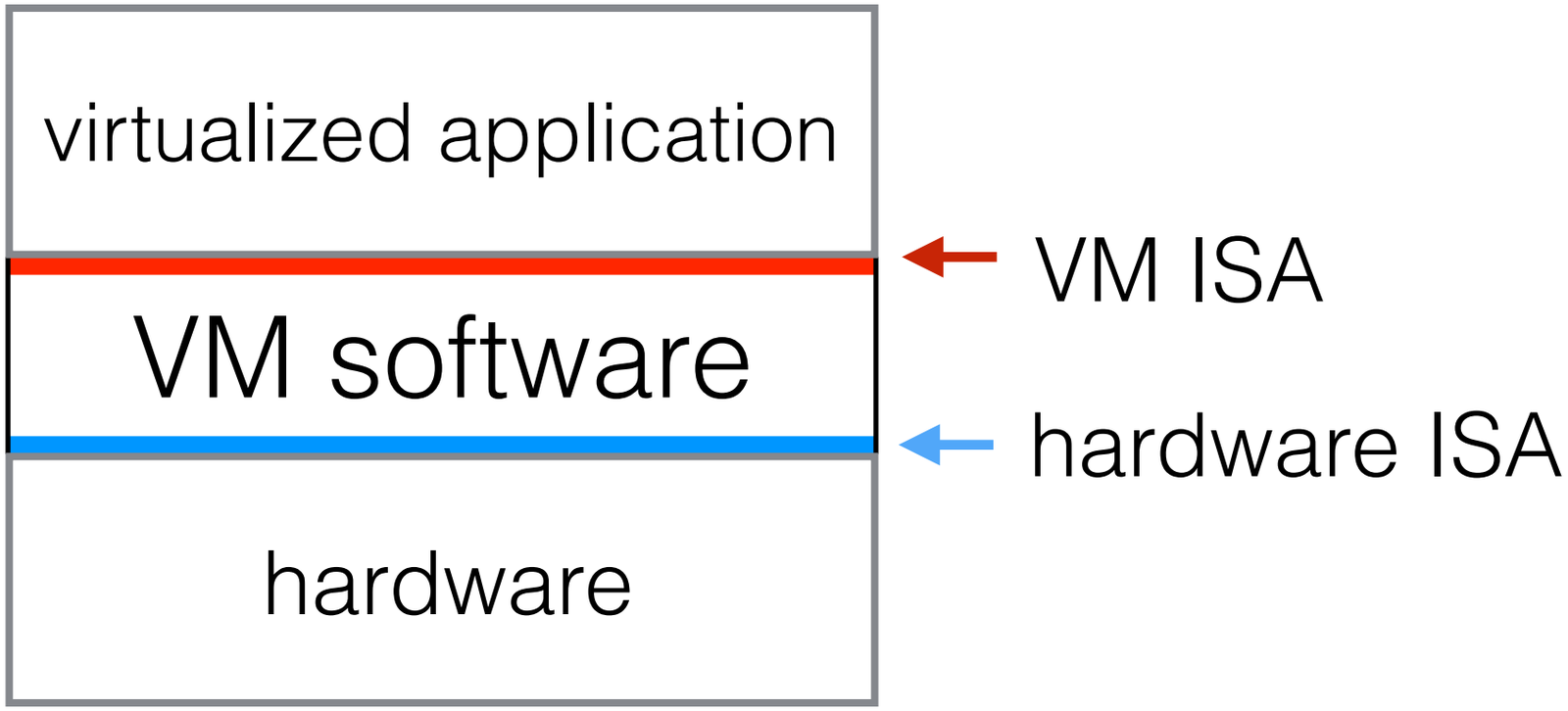
Generic VM architecture



What is a Virtual Machine?

- A software implementation of a machine architecture
  - Software needs hardware to run, so hardware is implied too
- Two machine architectures are involved: *guest* and *host* (although they might be the same! — more later)
- The guest may be an abstract machine or an emulation of a real machine (i.e., also available as a hardware implementation)
- The host is usually hardware, but need not be (e.g., a language VM written in Java running on the JVM)

A typical VM


virtualized application
VM software
hardware
VM ISA
hardware ISA

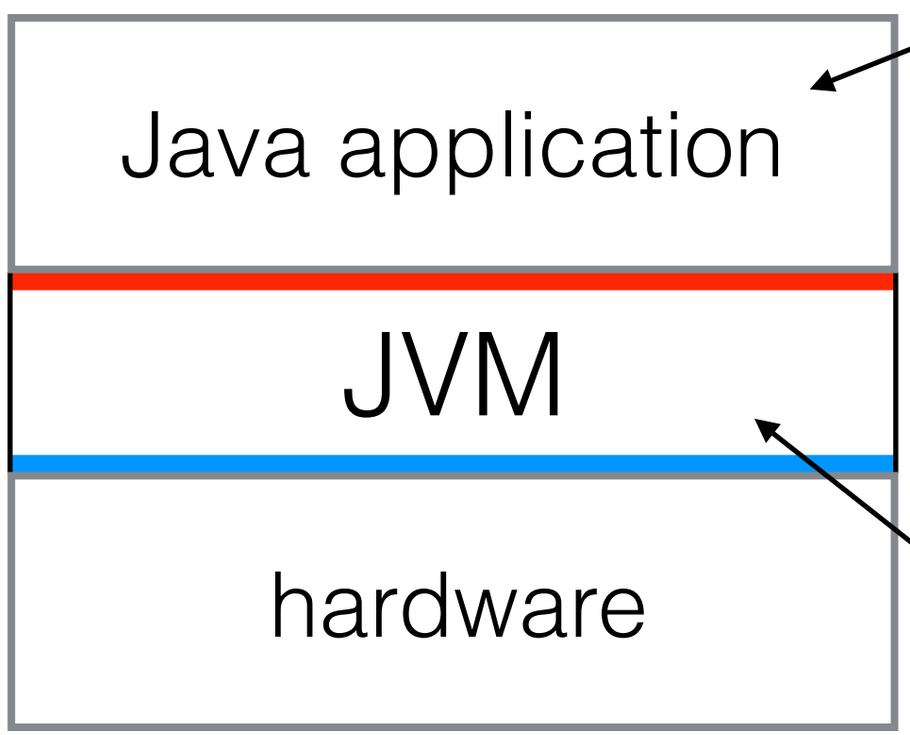
A smattering of VMs

- To illustrate the variety of VMs in past or present use:
  - The Java Virtual Machine — a language VM
  - VirtualBox — a hosted system VM
  - Rosetta — a binary translator
  - Dynamo — a dynamic binary optimizer
  - Crusoe — a co-designed hardware/software VM

Examples:
#1. The Java Virtual Machine (JVM)

- An example of a *language VM*, i.e., designed to run a single language (Java)
  - Has been repurposed for many other languages (Scala, Jython, Clojure, JRuby, JavaScript…see Wikipedia List_of_JVM_languages); some fit better than others
  - There are many other language VMs with a wide variety of sophistication in implementation, for: JavaScript, C#[?], Python, R, PHP, Ruby, Perl, Lua…
- First developed in the mid-'90s; reminiscent of earlier language VMs from the '80s (Pascal, Smalltalk)
- Massive adoption of Java spurred huge investment in VM implementations leading to transfer of research techniques to production, and development of many new approaches (especially for garbage collection and concurrency)
- Huge coverage
  - Deployed at a vast range of scales (from sensors through feature phones and smartphones, BluRay players and other appliances, tablets, laptops and desktops to servers and clusters)
- (Much more detail about the JVM architecture to follow in a later lecture.)

JVM



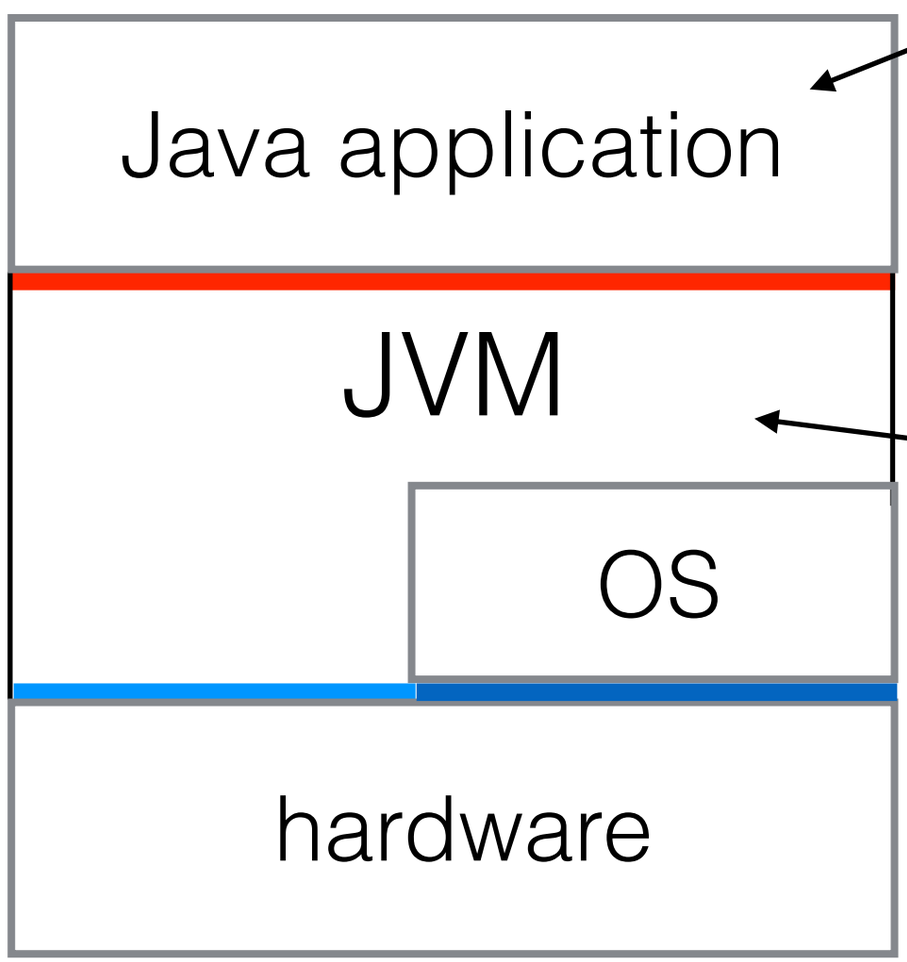
application written in Java, compiled to Java bytecode using *javac*

JVM ISA (bytecode)

hardware ISA

JVM is written in, e.g., C++ (e.g., HotSpot JVM) and compiled to the hardware ISA

Widespread approach: JVM as a process



application written in Java, compiled to Java bytecode using *javac*

JVM ISA (bytecode)

JVM with a single application runs as a single OS process

use OS to cooperate with hardware.

hardware ISA (user and privileged)

Aside: JVM concurrency

- Java was the first widely-adopted language with built-in concurrency
- The early JVMs ran on a single OS thread, multiplexing bytecodes (ie thread switch was done at bytecode boundaries, in software)
  - Sometimes known as “Green threads” (Green was the name of the underlying OS in the first Java implementation [which was then known as “Oak”]). The earlier (and more descriptive) term was *cooperative multithreading*.
- Current JVMs typically map Java threads onto OS threads.

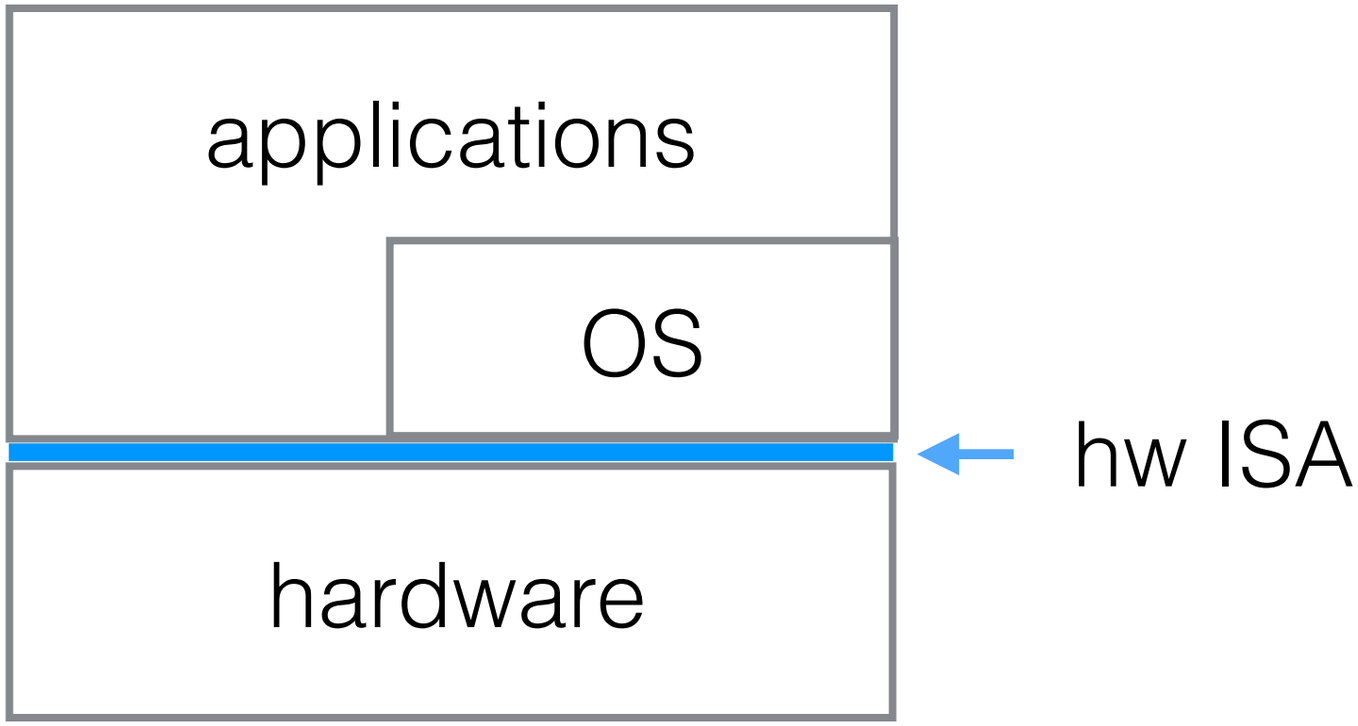
☆ used in exercise.

Example #2: VirtualBox

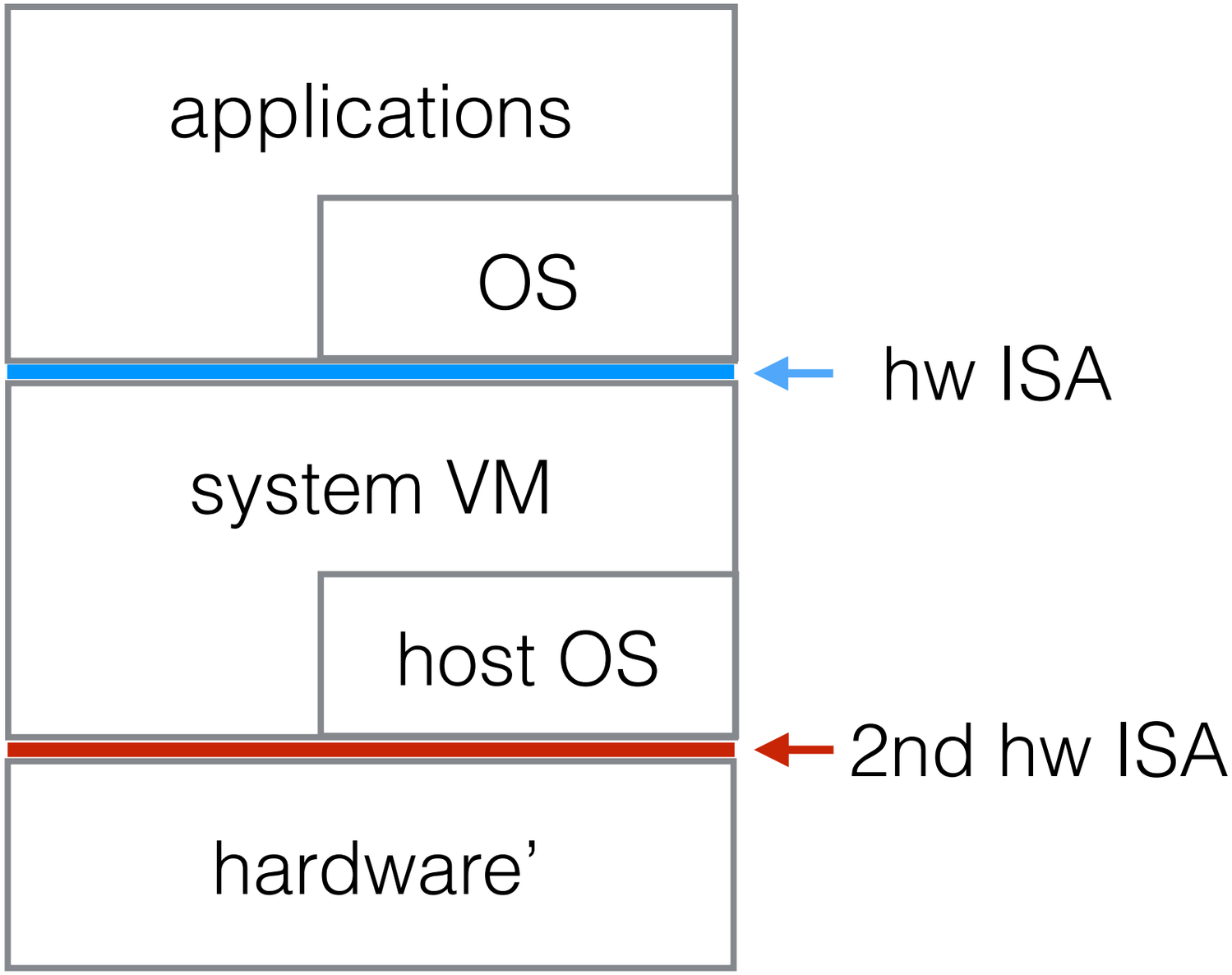
virtualbox.org

- Runs x86 systems (OS and applications) on x86. **Why?**
  - To run a different OS under the host OS
  - To run multiple guest systems simultaneously
- An example of a *system VM* and a *hosted VM* (TBD)
- Many other similar products: VMware (various), Parallels, QEMU, …
- You'll be using VirtualBox in the lab exercises.

Architecture of Hosted System VMs

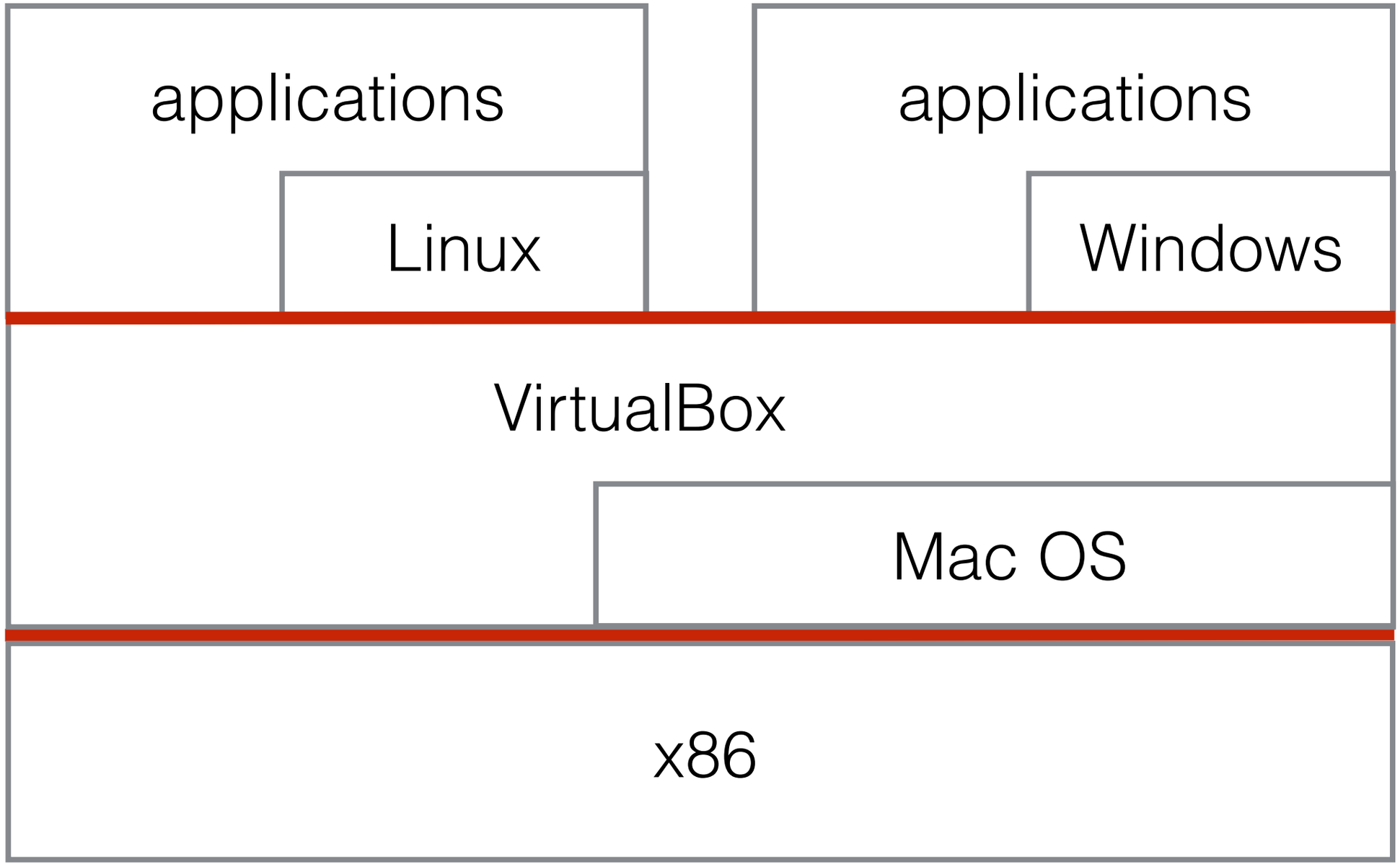


Architecture of Hosted System VMs



VM can translate application ISA to host ISA, which can run on its own hardware.

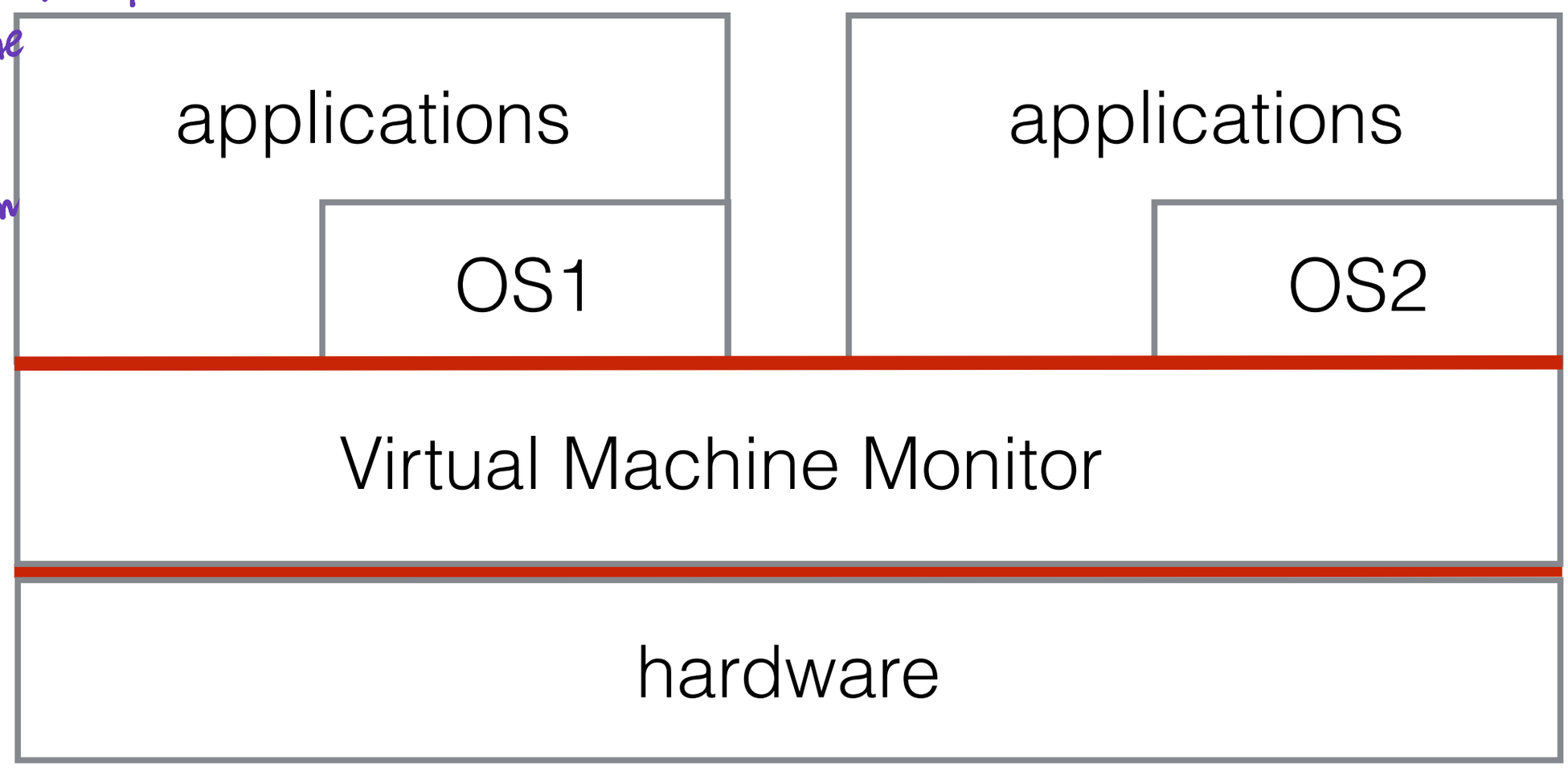
VirtualBox example



“Classic” System VM architecture (e.g., IBM System/370)

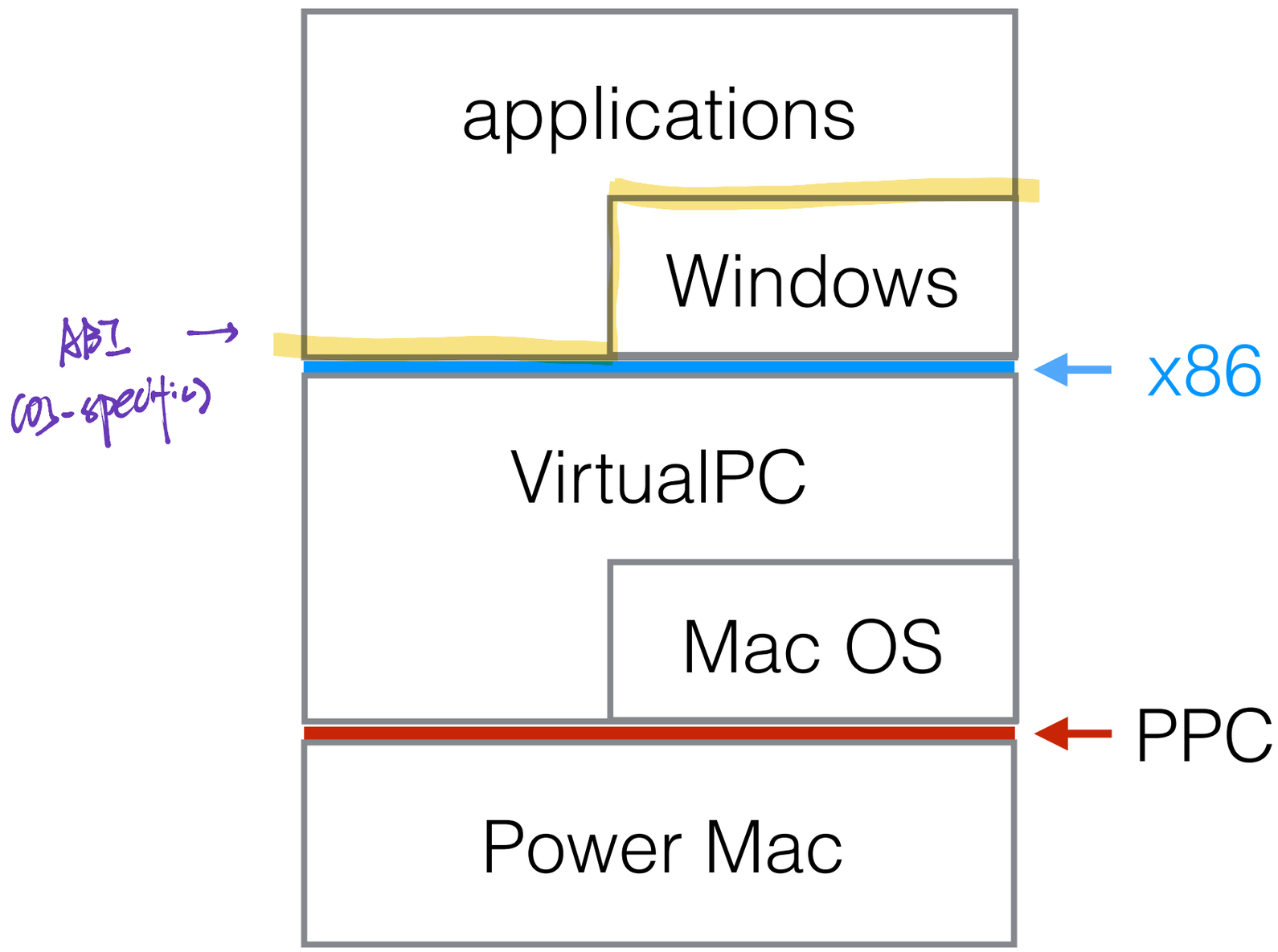
Hardware is very expensive.

IBM invents the monitor to run different application with their specific OS.



ISAs are the same, but there is no host OS

A Whole-System VM: *Virtual PC* (1997)



system VM → run a full computer inside the computer (enumerate hardware)

process VM → run a specific program in an isolated, platform-independent environment.

Definitions

Comes from book.

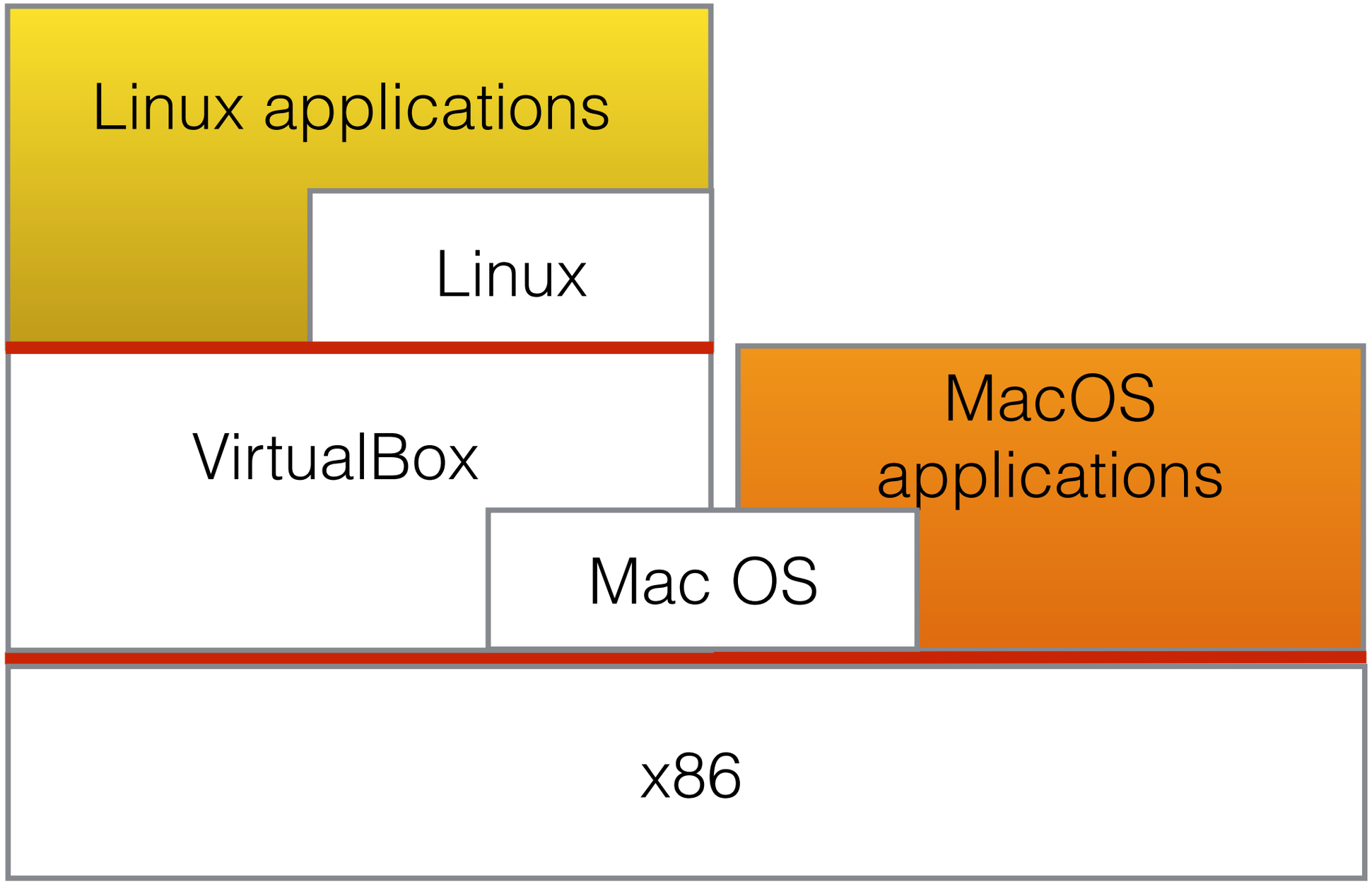
- A **Process VM** implements an **ABI** (Application Binary Interface: the combination of a user-level ISA and an OS system call interface)

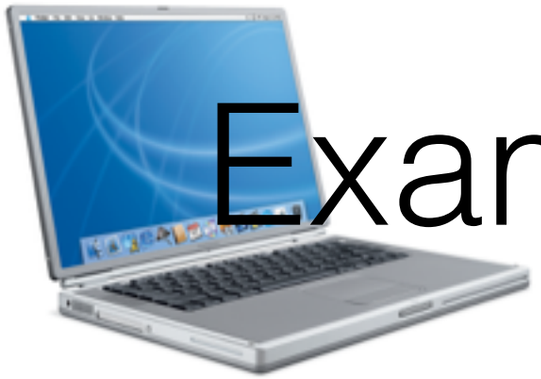
- A **System VM** implements *both* the hardware user and system ISA

  - A **Hosted System VM** has a host OS

  - A **Classic System VM** has a Virtual Machine Monitor (not a full OS; you can't run applications on the VMM as it does not implement the ABI)

Hosted VM — two levels for applications





Example #3: Apple's *Rosetta*

Binary translator

- 2005: Used on newly-introduced x86 MacBooks to emulate the PowerPC architecture of the earlier PowerBooks.

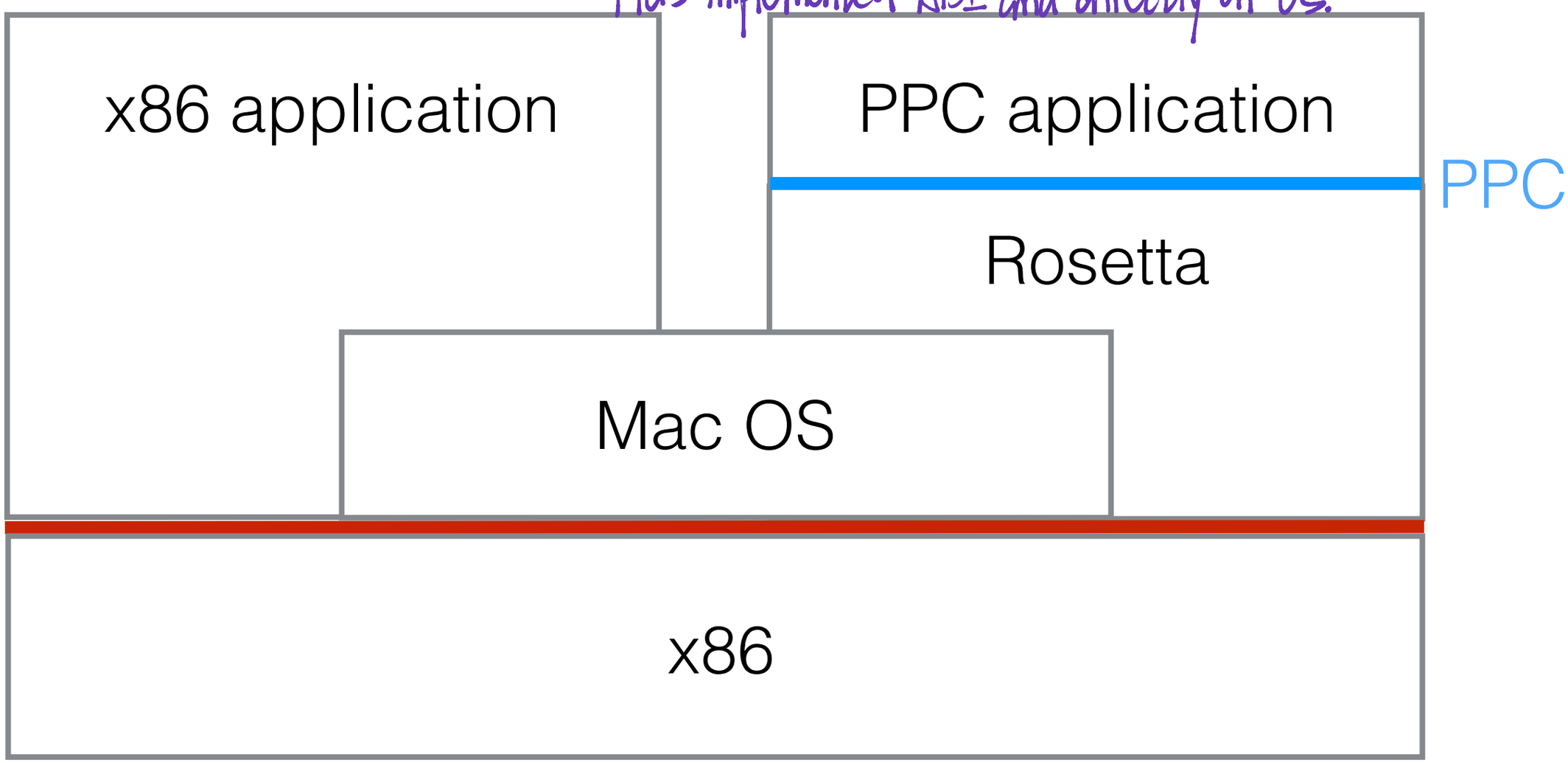
- Ensured the installed base of PPC applications would run on the new architecture, before they were ported

- An instantiation of the QuickTransit binary translator developed by Transitive Technologies in the UK (company acquired by IBM in 2009).

Rosetta architecture

(process VM)

Has implemented ABI and directly on OS.



How it worked

- When a PowerPC binary was invoked, the Rosetta layer performed a combination of interpretation of PPC instructions and dynamic binary translation (translating chunks of PPC code to x86) to run the application

- Because the new x86 processors performed better than the previous-generation PPC processors, the resulting performance was comparable, and therefore “good enough” until the applications were available as x86 binaries

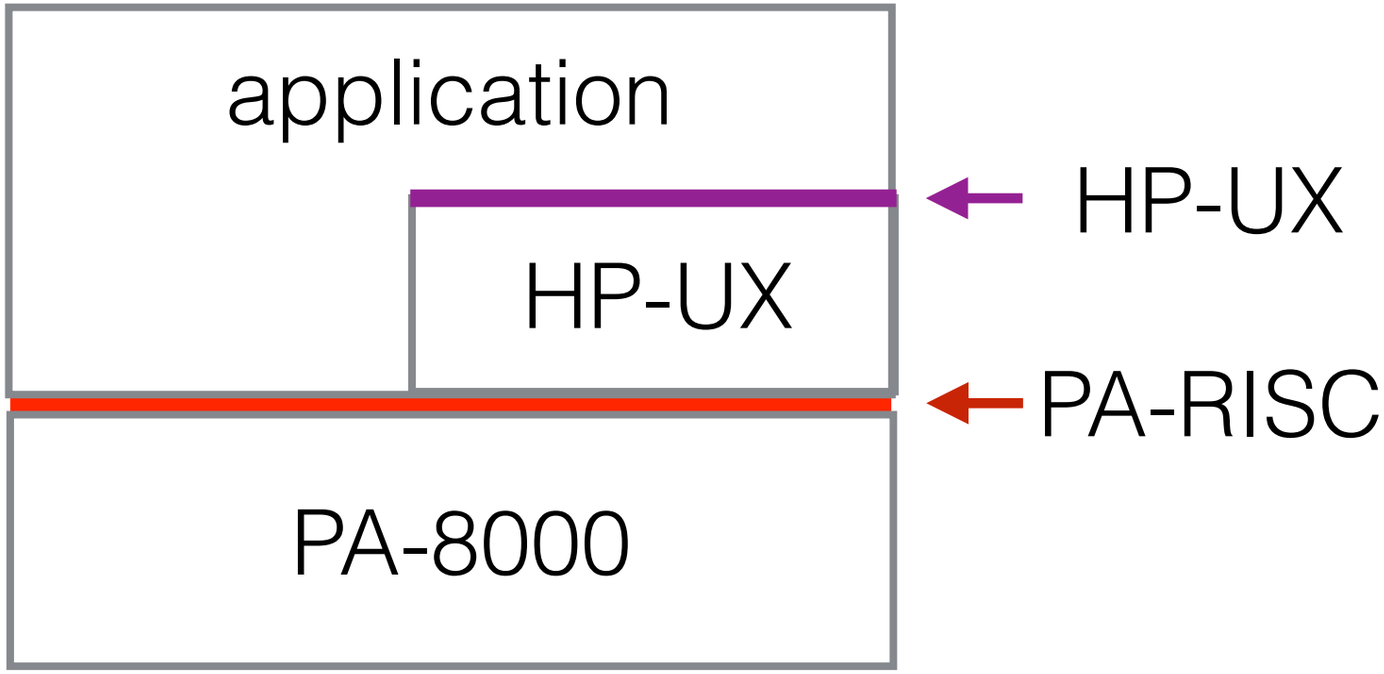
- This was the *second* such system deployed by Apple: they used a dynamic binary translator when moving from 68K to PPC (Wikipedia: Mac_68k_emulator).

Example #4: the *Dynamo* binary code optimizer (process)

The goal is to make it faster.

- Ran HP PA-8000 PA-RISC applications, dynamically re-optimizing hot code to improve performance
- An example of a **trace-driven** optimizer
  - The unit of optimization is an *instruction trace* (can span many basic blocks and procedures)
  - We will look at tracing compilers in a later lecture

Dynamo

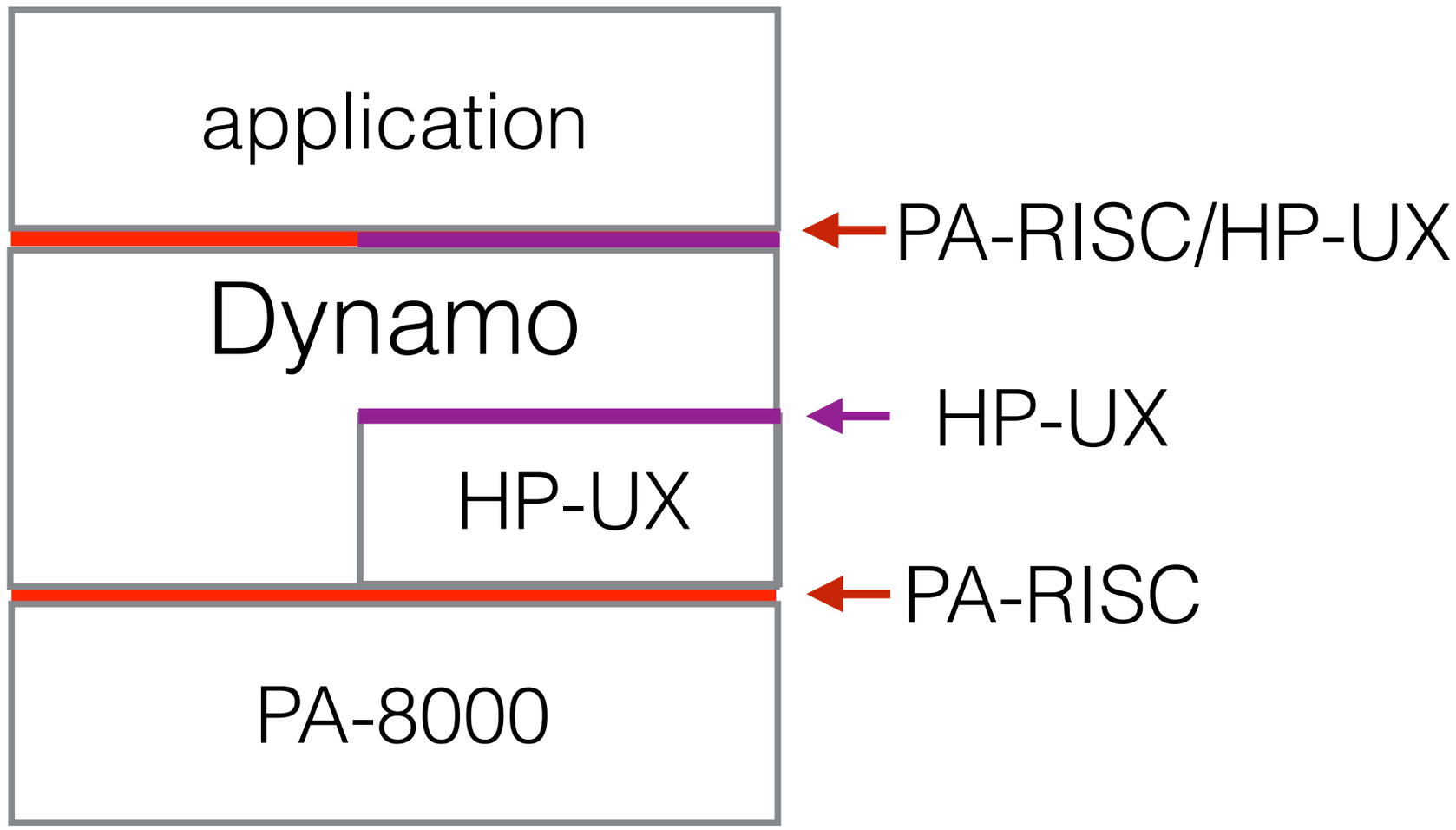


Dynamo

map original instruction

set to optimized

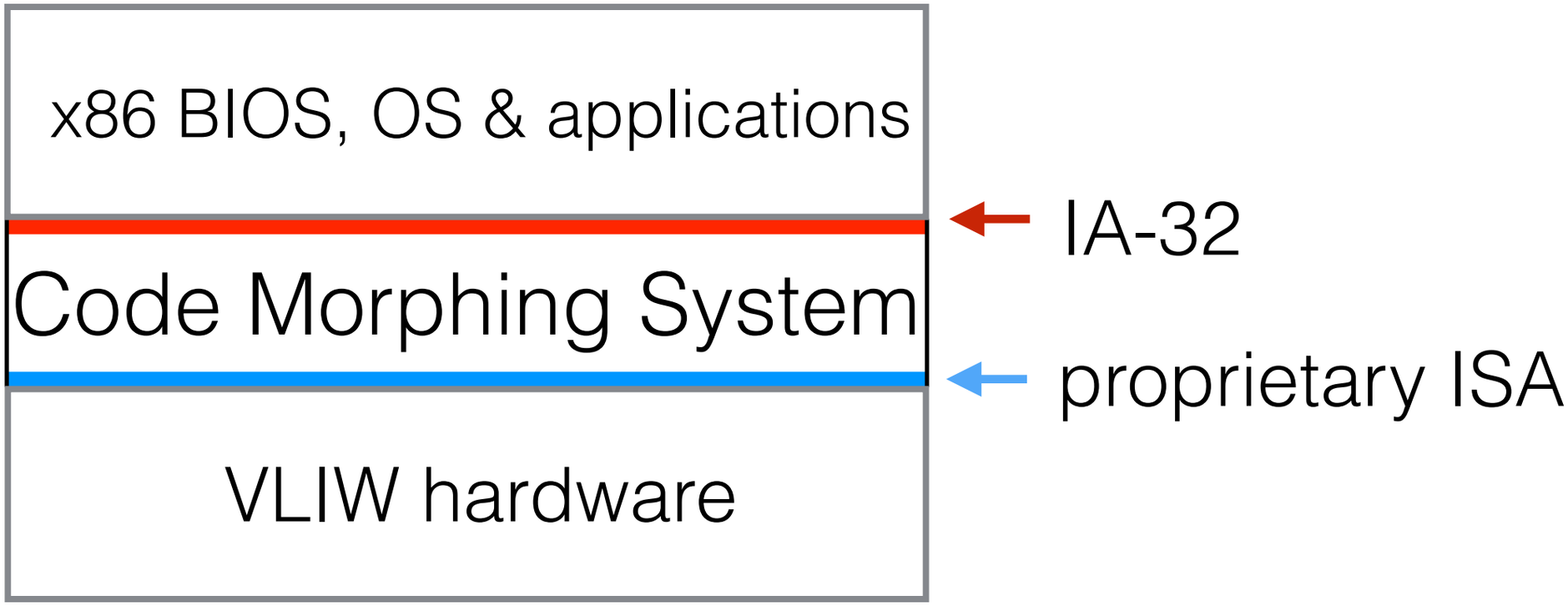
instruction set.



Example #5: Transmeta's *Crusoe*

- Co-designed software and hardware that ran IA-32 code on a VLIW-architecture microprocessor via a dynamic binary translator.
- Goal was improved power-efficiency with performance comparable to a conventional x86 implementation
  - Achieved by eliminating power-hungry hardware, such as fast x86 instruction decode
- Used in a variety of laptops, tablets, notebooks
- Introduced in 2000; a follow-on (Efficeon) in 2003

Crusoe system architecture



- CMS used many of the techniques we will cover later (feedback-driven optimization, speculation)
- Hardware provided additional support to deal with speculation and translation (e.g., shadow registers and gated store buffer with rollback triggered by alias detection)

Smith & Nair VM taxonomy

	Process VMs	System VMs
Same ISA	Dynamic binary optimizers	Classic System VMs Hosted VMs
Different ISA	Dynamic translators Language VMs	Whole-System VMs Co-designed VMs

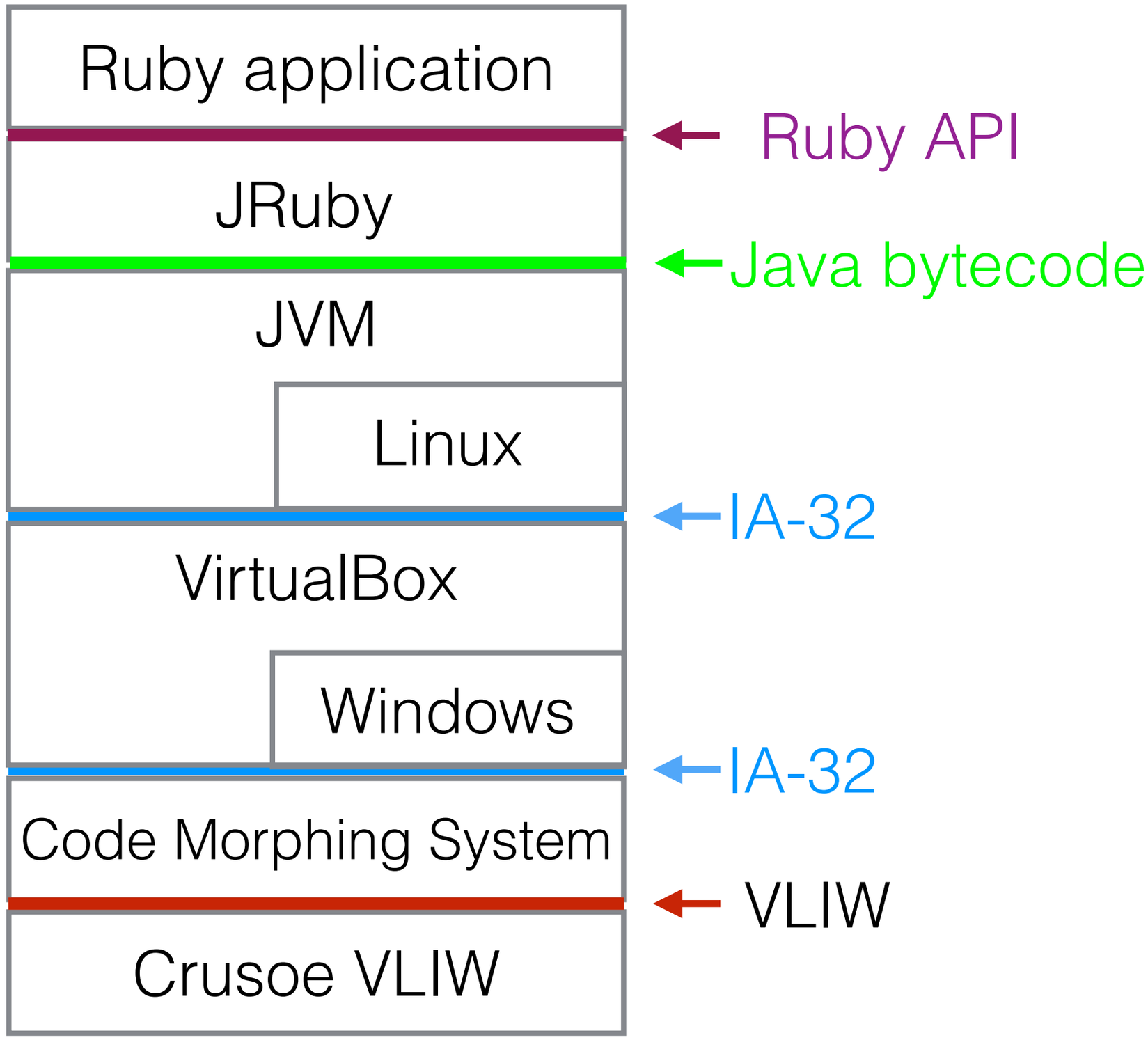
Some other terms that get thrown around

- **Emulator** can only involve memory contents!
  - A *machine emulator* typically mimics a real instruction set; sometimes just user mode
  - Not usually anywhere near as fast as the original system
  - Sometimes called a simulator; can omit detail (e.g., side effects, or a precise accounting of time), allow extra observation, attached microarchitecture models, simulated devices
  - Examples: QEMU; Simics (simulator)
  - Aside: Emulation is not limited to machines: e.g., CodeWeavers' *Crossover* emulates the Windows API on a Mac; an *in-circuit emulator* mimics the electrical interface

More terms

- Interpreter, JIT — TBD
- Virtual Machine Monitor, hypervisor
  - Hypervisor usually associated with partitioning (i.e., multiple guests)
- source and target — alternatives to guest and host
  - Often used in conjunction with binary translation; *source* is a bit misleading
- native ISA, virtual ISA — host and guest ISAs
  - The hardware ISA is the native ISA

VMs can be stacked



What is a managed runtime?

- A runtime for a managed language (duh)
- A compiler for a managed language produces **managed code**
- Managed code does not have unfettered hardware access: some layer of software intermediates
  - Could be a VM; but the format need not be an instruction set, e.g., source code (JavaScript), compressed ASTs
- The terms appear to have originated with Microsoft's Common Language Runtime
  - …which is a VM(!)
- I will use "Virtual Machine" more-or-less to cover all these systems

Why use a VM?

- Portability — decouples the guest from the host.
- Improved security, e.g., via a “sandbox”
- Virtualization of hardware (one piece of hardware can act as many): run two jdk on machine.
  - Consolidation; version management; partitioning of resources; ability to run different flavors of OS simultaneously
- Persistence via snapshots, (live) migration
- Instrumentation/observation
- Convenience: cost-saving and time-saving
  - Don’t have to procure a new system; easier to code for
- Performance — sometimes — with less effort

Score card example: VirtualBox

- ✓ Portability — run Linux applications on Windows, for example
- ✓ Improved security via a “sandbox” — Guest app has managed access to host filesystem
- ✓ Virtualization of hardware (one piece of hardware can act as many).
  - Consolidation; version management; partitioning of resources; ability to run different flavors of OS
- ✓ Persistence via snapshots, (live) migration
- ✓ Instrumentation/observation — can monitor virtualized I/O devices (e.g., display)
- ✓ Convenience: cost-saving and time-saving
  - Don’t have to procure a new system to run a second OS
- ✘ Performance

Score card example: JVM

- ✓ Portability — “Write Once, Run Anywhere”
- ✓ Improved security via a “sandbox”
- ✘ Virtualization of hardware (one piece of hardware can act as many).
- ✘ Persistence via snapshots, (live) migration
  - ✓ But other language VMs offer this
- ✓ Instrumentation/observation — e.g., can debug an application without compiling a special version
- ✓ Convenience: cost-saving and time-saving — WORA, again
- ✓ Performance — with less effort

What’s next, part 1

- Introduction to Feeny (our toy language) by Patrick Li — exercise 1
- AST interpretation — exercise 2
- Next week: VM internals, bytecodes

Later…

- Dissection of the JVM spec.
- Memory management (in two acts)
- Just-in-time compilation
- Dynamic optimizing compilation
- Using a VM construction kit

And finally…

- Tracing and meta-tracing compilation
- System VMs
- Tools — debugging, profiling, etc.

FAQ #5. Why the emphasis on language VMs (at the expense of system VMs)?

- Everyone can invent a language and/or build a VM for one; hardly anyone gets to invent a native ISA that matters.
- The cross-product of native ISAs that matters hardly ever changes.
- Most of the complexity of system VMs is ugly detail; not hard, just unpleasant, e.g., self-modifying code
- Many of the underlying techniques are common
- The first implementation of a native ISA is in silicon, and a VM is unlikely to make it cheaper or more efficient
- Language VMs create an opportunity; system VMs patch over a problem (co-designed VMs being the counter-example)