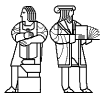
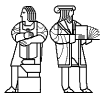


Virtual Machines and Dynamic Translation: Implementing ISAs in Software

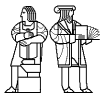
Krste Asanovic
Laboratory for Computer Science
Massachusetts Institute of Technology



Software Applications

How is a software application encoded?

- What are you getting when you buy a software application?
- What machines will it work on?
- Who do you blame if it doesn't work, i.e., what contract(s) were violated?

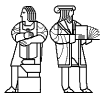


ISA + Environment = Virtual Machine

ISA alone not sufficient to write useful programs, need I/O

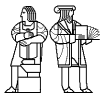
- **Direct access to memory mapped I/O via load/store instructions problematic**
  - **time-shared systems**
  - **portability**
- **Operating system responsible for I/O**
  - **sharing devices and managing security**
  - **hiding different types of hardware (e.g., EIDE vs. SCSI disks)**
- **ISA communicates with operating system through some standard mechanism, i.e., `syscall` instructions**
  - **example convention to open 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
```



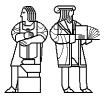
Application Binary Interface (ABI)

- Programs are usually distributed in a binary format that encode the program text (instructions) and initial values of some data segments
- Virtual machine specifications include
  - what state is available at process creation
  - which instructions are available (the ISA)
  - what system calls are possible (I/O, or the *environment*)
- The ABI is a specification of the binary format used to encode programs for a virtual machine
- Operating system implements the virtual machine
  - at process startup, OS reads the binary program, creates an environment for it, then begins to execute the code, handling traps for I/O calls, emulation, etc.



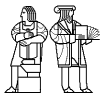
OS Can Support Multiple VMs

- Virtual machine features change over time with new versions of operating system
  - new ISA instructions added
  - new types of I/O are added (e.g., asynchronous file I/O)
- Common to provide backwards compatibility so old binaries run on new OS
  - SunOS 5 (System V Release 4 Unix, Solaris) can run binaries compiled for SunOS4 (BSD-style Unix)
  - Windows 98 runs MS-DOS programs
- If ABI needs instructions not supported by native hardware, OS can provide in software



Supporting Multiple OSs on Same Hardware

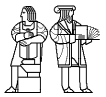
- Can virtualize the environment that an operating system sees, an OS-level VM
- Hypervisor layer implements sharing of real hardware resources by multiple OS VMs that each think they have a complete copy of the machine
  - Popular in early days to allow mainframe to be shared by multiple groups developing OS code
  - 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 state that OS needs
  - easier if OS interface to hardware well defined



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 instructions in MicroVax 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

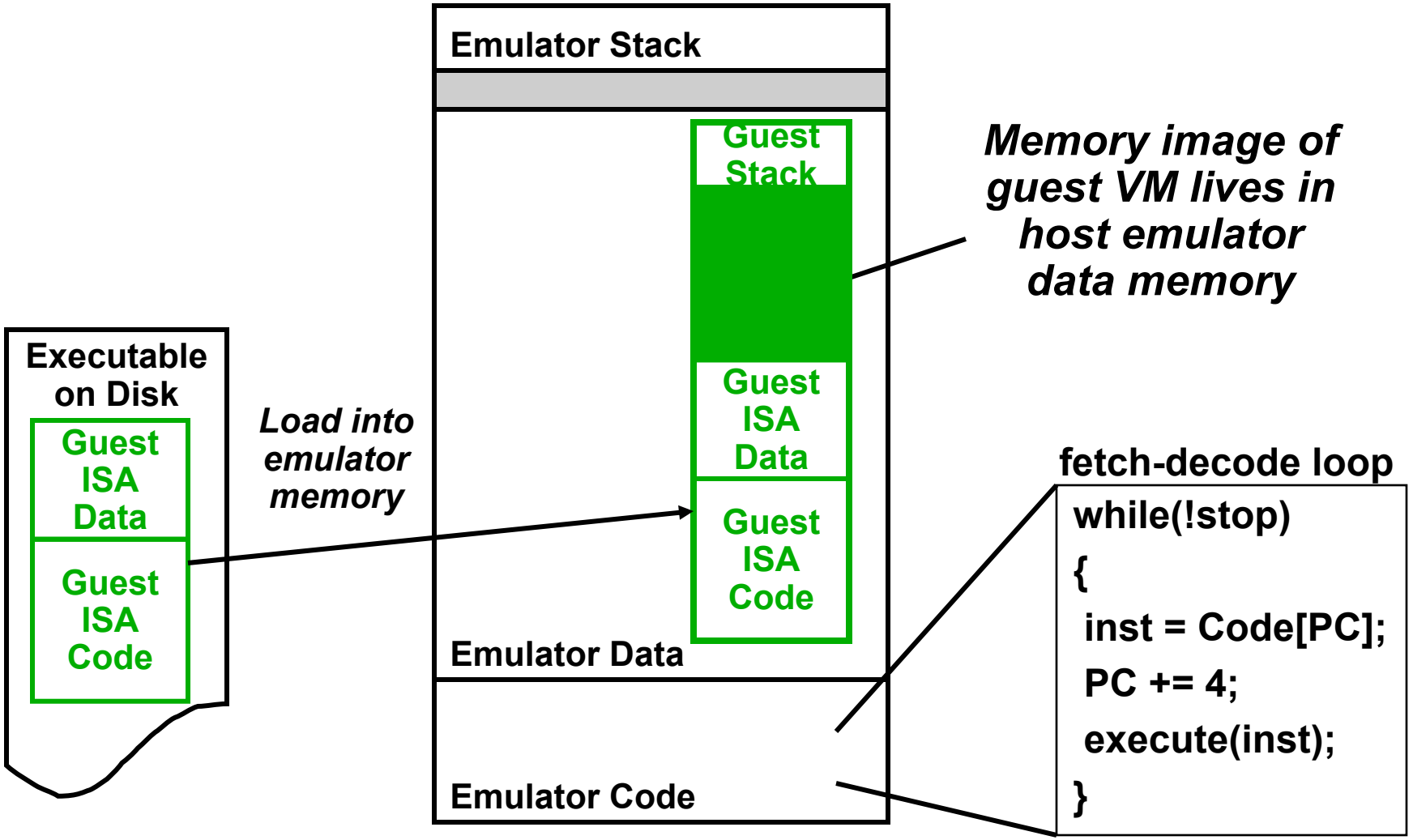
Run programs for one ISA on hardware with different ISA

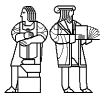
Techniques:

- **Emulation**
  - **OS software interprets instructions at run-time**
  - **E.g., OS for PowerPC Macs had emulator for 68000 code**
- **Binary Translation**
  - **convert at install and/or load time**
  - **IBM AS/400 to modified PowerPC cores**
  - **DEC tools for VAX->MIPS->Alpha**
- **Dynamic Translation (or Dynamic Compilation)**
  - **compile non-native ISA to native ISA at run time**
  - **Sun's HotSpot Java JIT (just-in-time) compiler**
  - **Transmeta Crusoe, x86->VLIW code morphing**
- **Run-time Hardware Emulation**
  - **Hardware supports two ISAs!**
  - **IBM 360 had IBM 1401 emulator in microcode**
  - **Intel Itanium converts x86 to native VLIW (two software-visible ISAs)**

Emulation

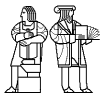
- Software instruction set interpreter fetches and decodes one instruction at a time in emulated VM





Emulation

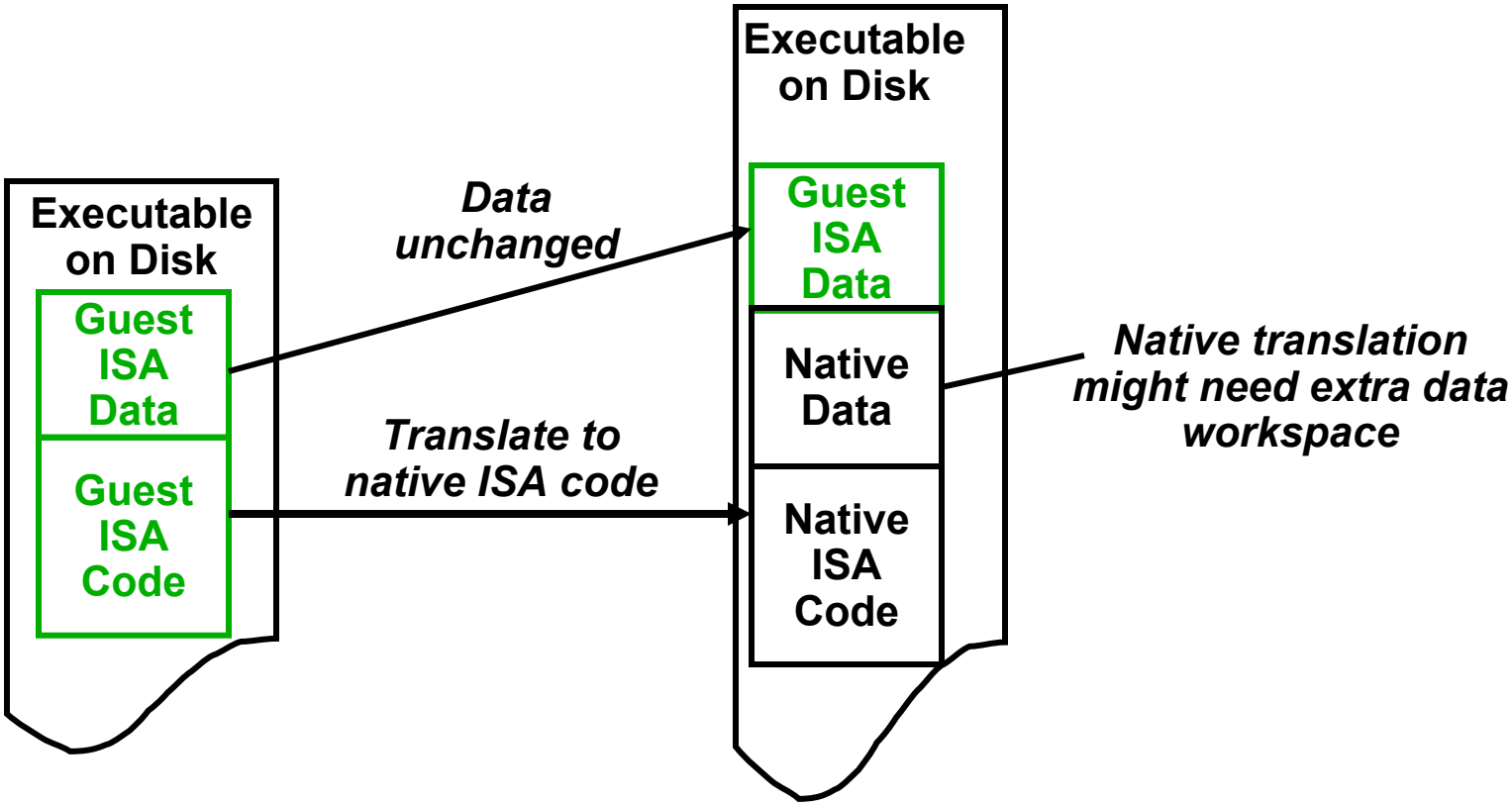
- Easy to code, small code footprint
- *Slow*, approximately 100x slower than native execution for RISC ISA hosted on RISC ISA
- Problem is time taken to decode instructions
  - fetch instruction from memory
  - switch tables to decode opcodes
  - extract register specifiers using bit shifts
  - access register file data structure
  - execute operation
  - return to main fetch loop



Binary 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 native ISA executable
- Removes interpretive fetch-decode overhead
- Can do compiler optimizations on translated code to improve performance
  - register allocation for values flowing between guest ISA instructions
  - native instruction scheduling to improve performance
  - remove unreachable code
  - inline assembly procedures

Binary Translation, Take 1



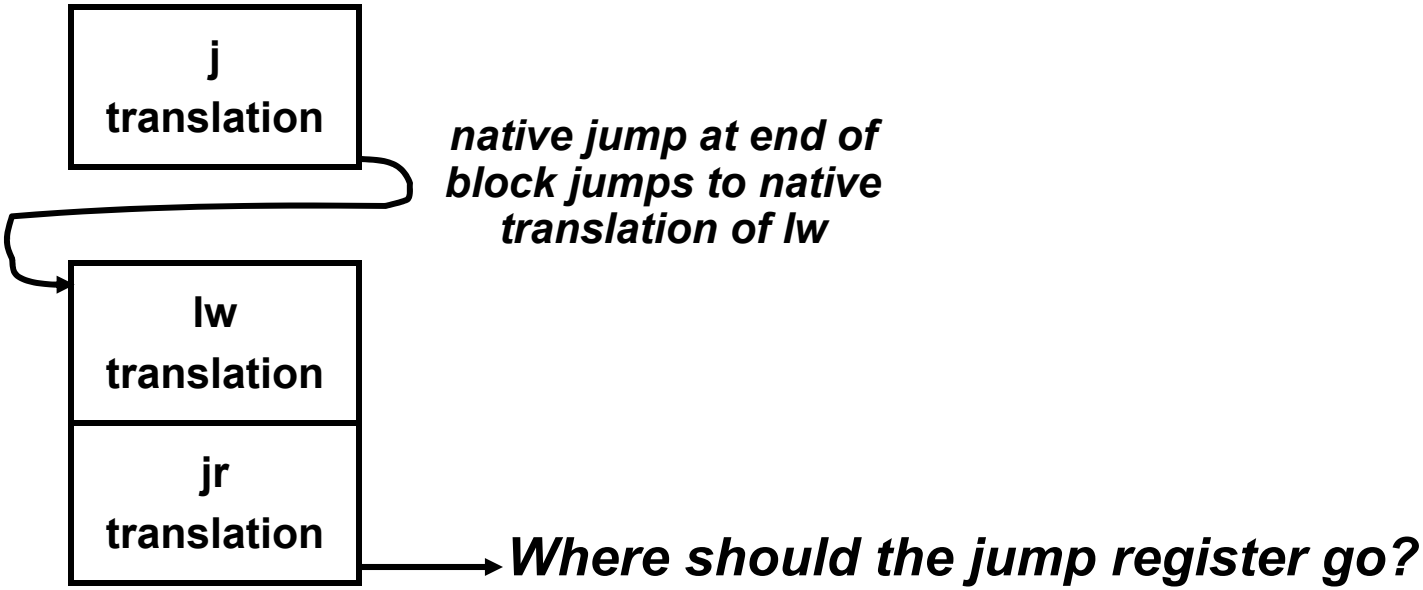
Binary Translation Problems

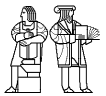
Branch and Jump targets

- guest code:

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

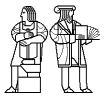
- native code





PC Mapping Table

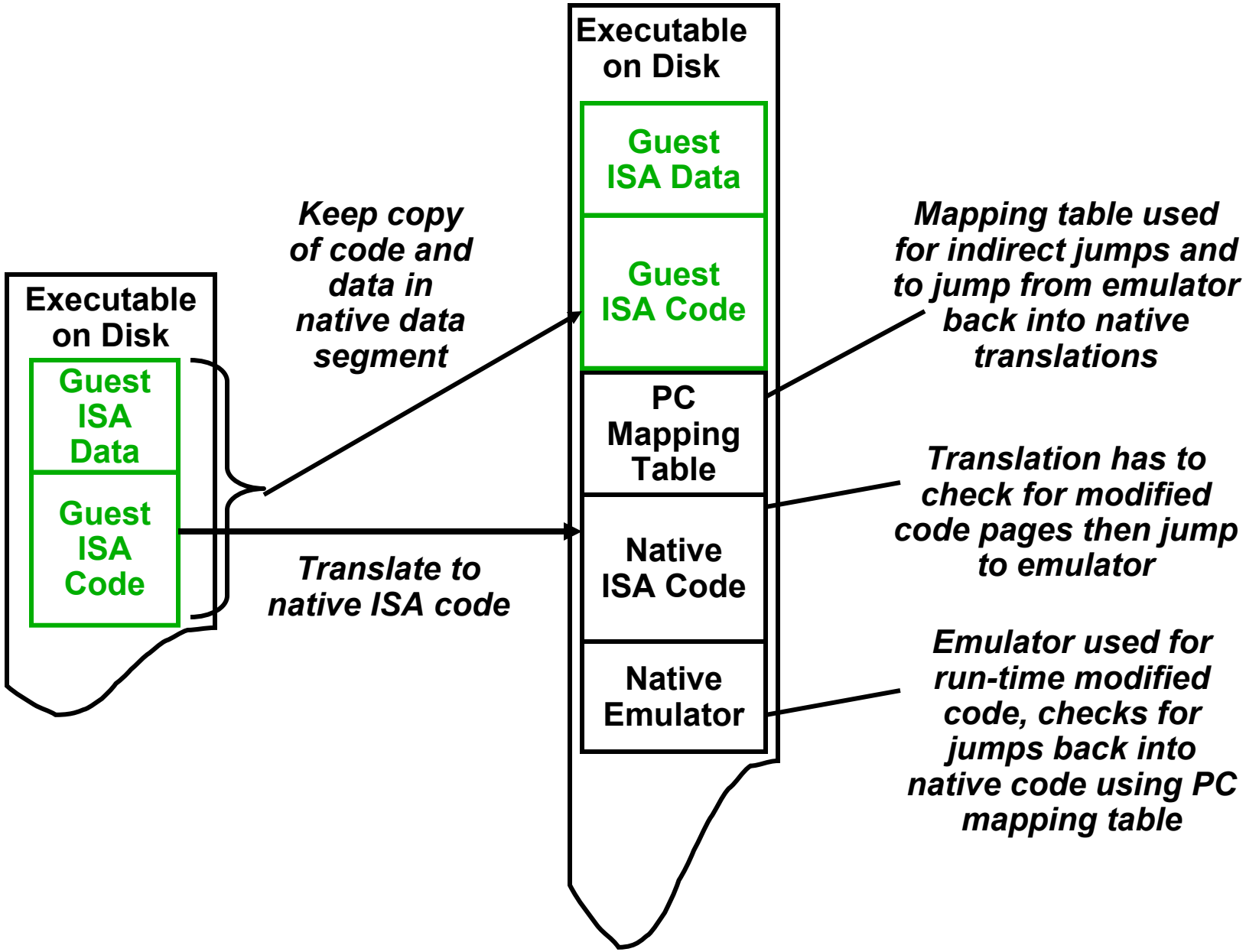
- Table gives translated PC for each hosted PC
- Indirect jumps translated into code that looks in table to find where to jump to
  - can optimize well-behaved guest code for subroutine call/return by using native PC in return links
- If can branch to any guest PC, then need one table entry for every 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
- Only minority of guest instructions are indirect jump targets, want to find these
  - highly structured VM design
  - run-time feedback of where targets were



Binary Translation Problems

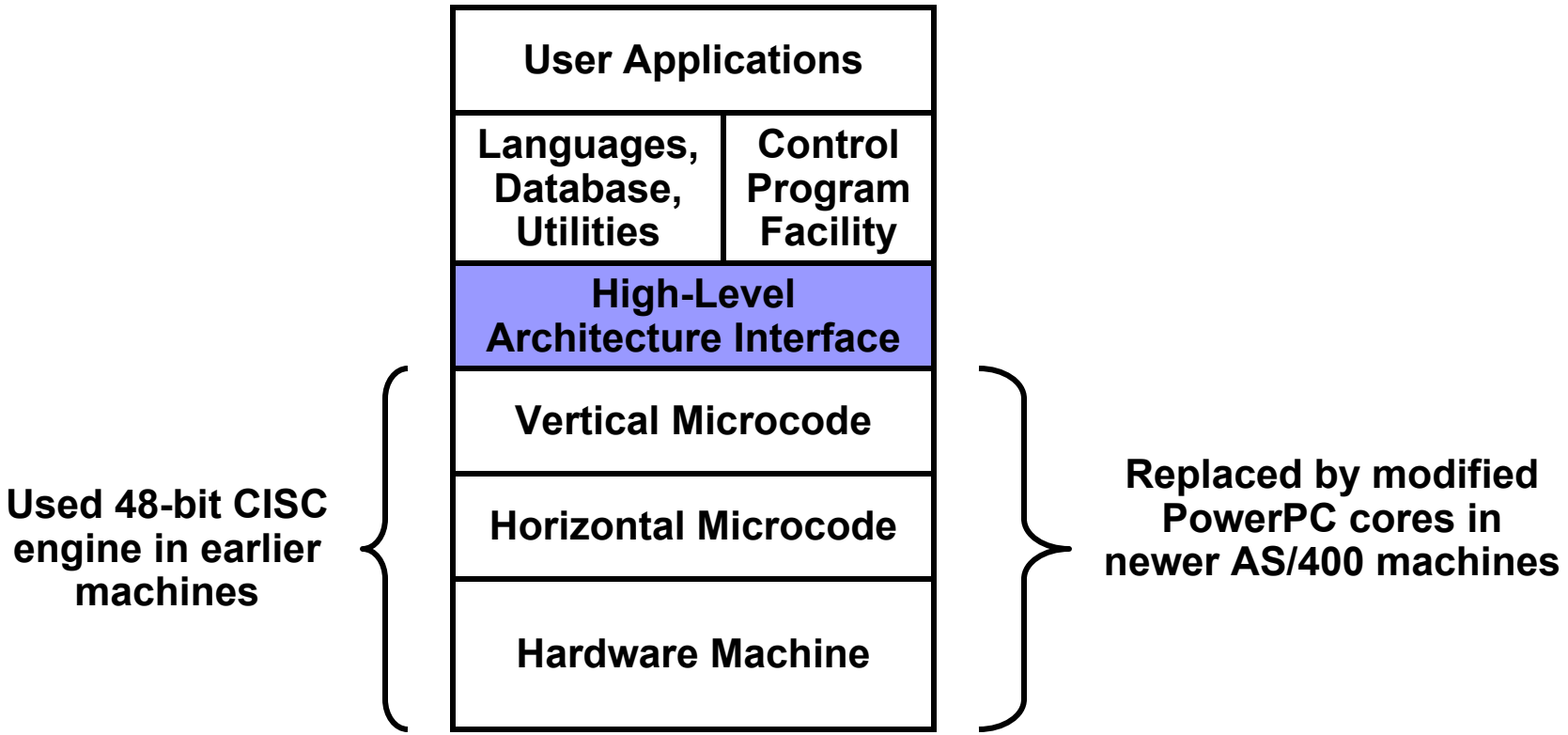
- **Self-modifying code!**
  - `sw r1, (r2)    # r2 points into code space`
- **Rare in most code, but has to be handled if allowed by guest ISA**
- **Usually handled by including interpreter and marking modified code pages as “interpret only”**
- **Have to invalidate all native branches into modified code pages**

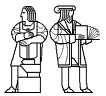
Binary Translation, Take 2



IBM System/38 and AS/400

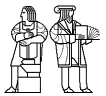
- System/38 announced 1978, AS/400 is follow-on line
- High-level instruction set interface designed for binary translation
- Memory-memory style instruction set, never directly executed by hardware





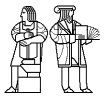
Dynamic Translation

- Translate code sequences as needed at run-time, but cache results
- 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
- Technique used in Java JIT (Just-In-Time) compilers
- Also, Transmeta Crusoe for x86 emulation



Transmeta Crusoe

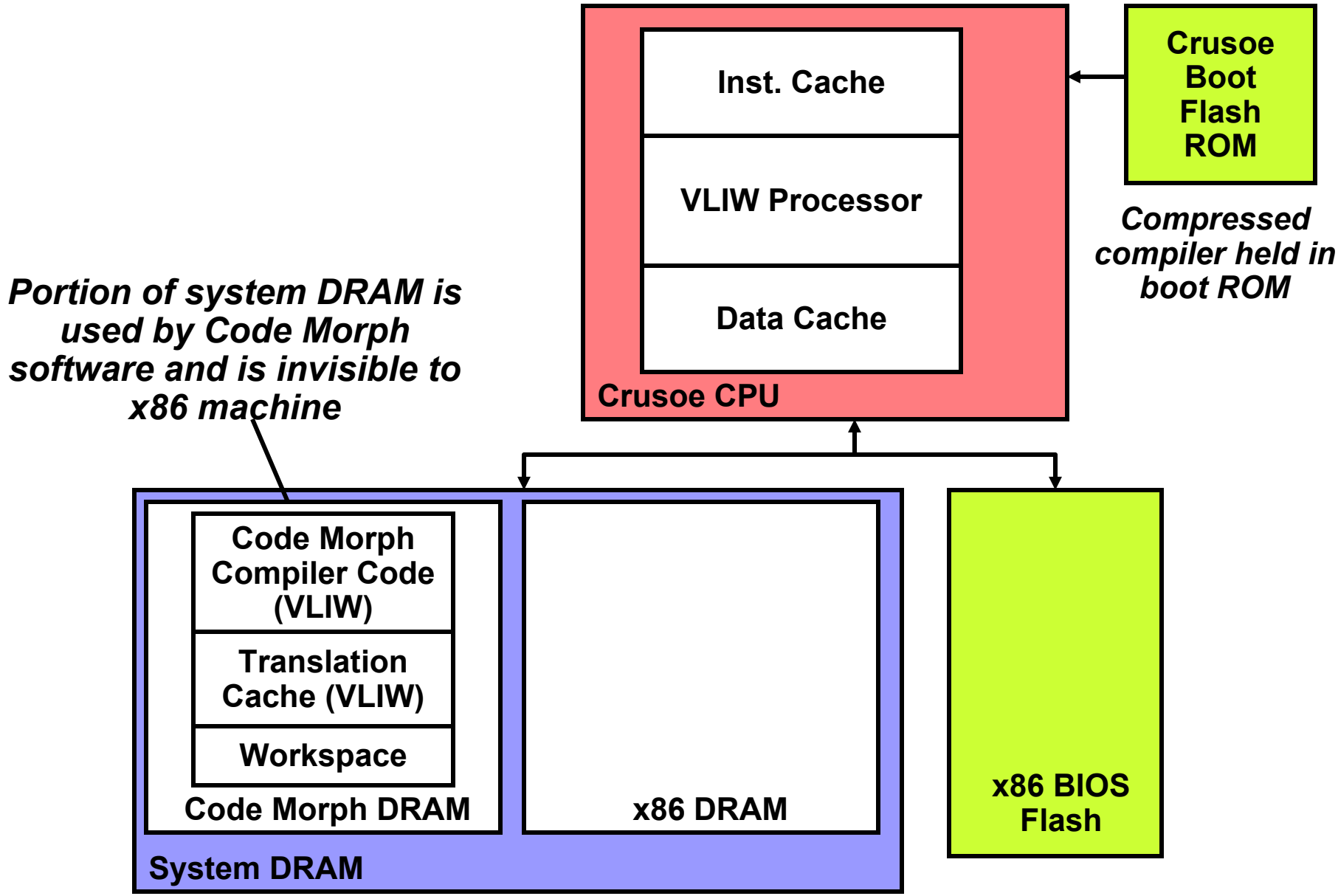
- Converts x86 ISA into internal native VLIW format using software at run-time → "Code Morphing"
- Optimizes across x86 instruction boundaries to improve performance
- Translations cached to avoid translator overhead on repeated execution
- Completely invisible to operating system – looks like x86 hardware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

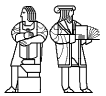


Transmeta VLIW Engine

- Two VLIW formats, 64-bit and 128-bit, contains 2 or 4 RISC-like operations
- VLIW engine optimized for x86 code emulation
  - evaluates condition codes the same way as x86
  - has 80-bit floating-point unit
  - partial register writes (update 8 bits in 32 bit register)
- Support for fast instruction writes
  - run-time code generation important
- Two different VLIW implementations, low-end TM3120, high-end TM5400
  - native ISA differences invisible to user, hidden by translation system

Crusoe System





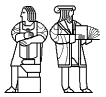
Transmeta Translation

x86 code:

```
addl %eax, (%esp) # load data from stack, add to eax
addl %ebx, (%esp) # load data from stack, add to ebx
movl %esi, (%ebp) # load esi from memory
subl %ecx, 5      # sub 5 from ecx
```

first step, translate into RISC ops:

```
ld %r30, [%esp]          # load from stack into temp
add.c %eax, %eax, %r30 # add to %eax, set cond.codes
ld %r31, [%esp]
add.c %ebx, %ebx, %r31
ld %esi, [%ebp]
sub.c %ecx, %ecx, 5
```



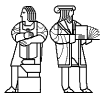
Compiler Optimizations

RISC ops:

```
ld %r30, [%esp]          # load from stack into temp
add.c %eax, %eax, %r30 # add to %eax, set cond.codes
ld %r31, [%esp]
add.c %ebx, %ebx, %r31
ld %esi, [%ebp]
sub.c %ecx, %ecx, 5
```

Optimize:

```
ld %r30, [%esp]          # load from stack only once
add %eax, %eax, %r30
add %ebx, %ebx, %r30    # reuse data loaded earlier
ld %esi, [%ebp]
sub.c %ecx, %ecx, 5     # only this cond. code needed
```



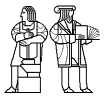
Scheduling

Optimized RISC ops:

```
ld %r30, [%esp]          # load from stack only once
add %eax, %eax, %r30
add %ebx, %ebx, %r30     # reuse data loaded earlier
ld %esi, [%ebp]
sub.c %ecx, %ecx, 5      # only this cond. code needed
```

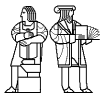
Schedule into VLIW code:

```
ld %r30, [%esp]; sub.c %ecx, %ecx, 5
ld %esi, [%ebp]; add %eax, %eax, %r30; add %ebx, %ebx, %r30
```



Translation Overhead

- Highly optimizing compiler takes considerable time to run, adds run-time overhead
- Only worth doing for frequently executed code
- Translation adds instrumentation into translations that counts how often code executed, and which way branches usually go
- As count for a block increases, higher optimization levels are invoked on that code



Exceptions

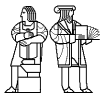
Original x86 code:

```
addl %eax, (%esp) # load data from stack, add to eax
addl %ebx, (%esp) # load data from stack, add to ebx
movl %esi, (%ebp) # load esi from memory
subl %ecx, 5      # sub 5 from ecx
```

Scheduled VLIW code:

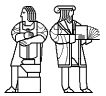
```
ld %r30, [%esp]; sub.c %ecx, %ecx, 5
ld %esi, [%ebp]; add %eax, %eax, %r30; add %ebx, %ebx, %r30
```

- **x86 instructions executed out-of-order with respect to original program flow**
- **Need to restore state for precise traps**



Shadow Registers and Store Buffer

- All registers have working copy and shadow copy
- Stores held in software controlled store buffer, loads can snoop
- At end of translation block, commit changes by copying values from working regs to shadow regs, and by releasing stores in store buffer
- On exception, re-execute x86 code using interpreter



Handling Self-Modifying Code

- When a translation is made, mark the associated x86 code page as being translated in page table
- Store to translated code page causes trap, and associated translations are invalidated