Virtual Machines: Architectures, Implementations and Applications

HOTCHIPS 17
Tutorial 1, Part 1

J. E. Smith
University of Wisconsin-Madison

Rich Uhlig
Intel Corporation

August 14, 2005

INTRODUCTION

Introduction

Why are virtual machines interesting?

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

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

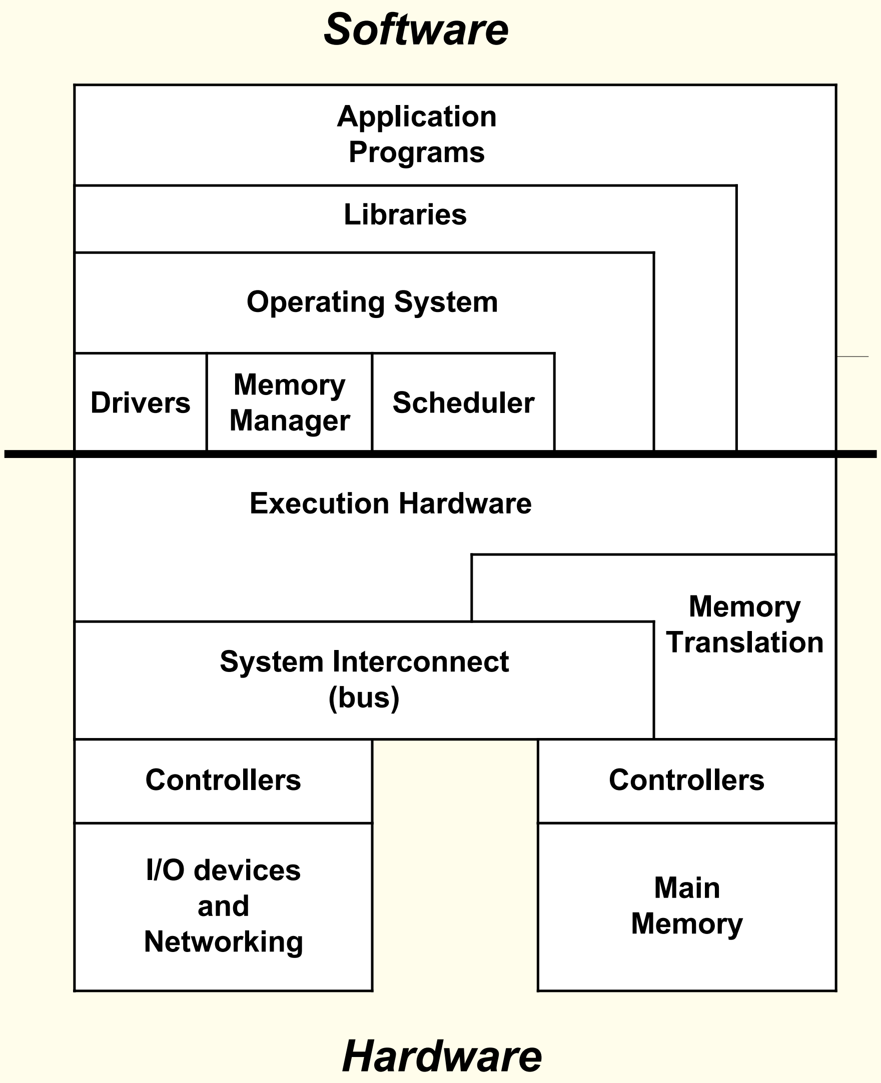
They involve computer architecture in a pure sense

Virtualization will be a key part of future computer systems

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

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

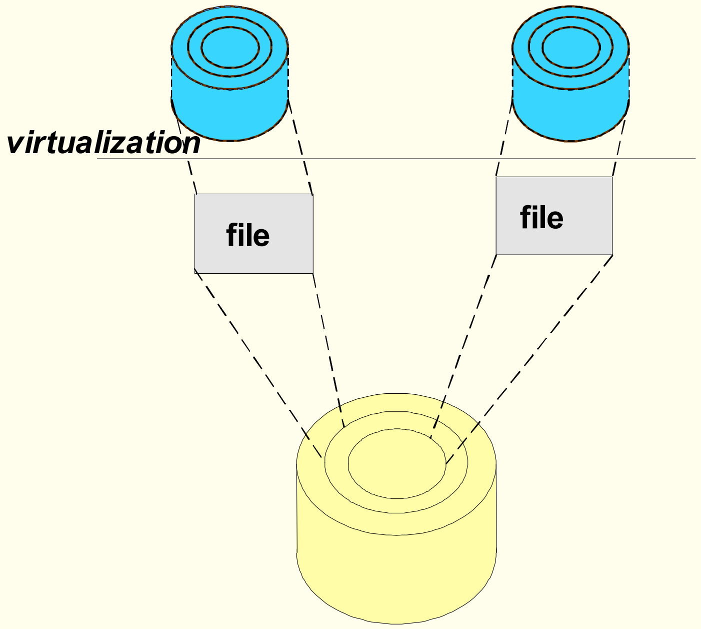
- ❑ **Similar to abstraction**

  Except

  - Details not necessarily hidden

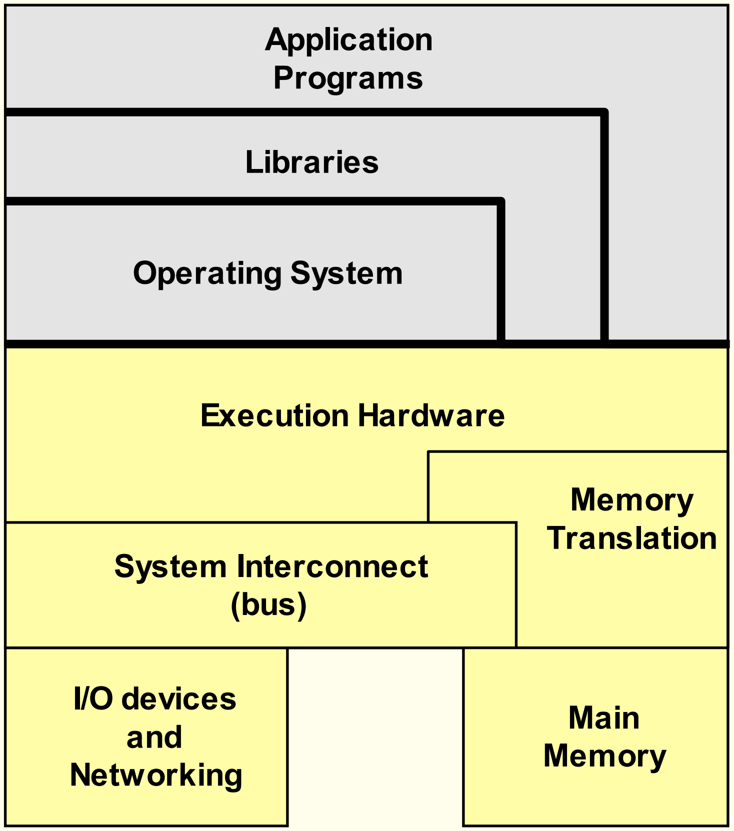
- ❑ **Construct Virtual Disks**
  - As files on a larger disk
  - Map state
  - Implement functions

- ❑ **VMs: do the same thing with the whole "machine"**



The “Machine”

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

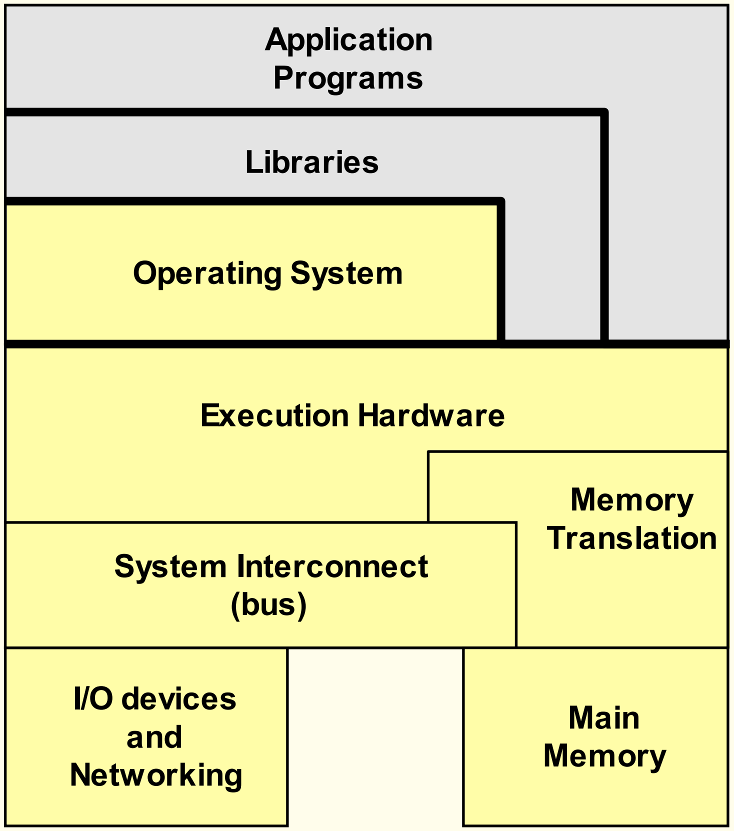


Instruction Set Architecture

- ISA
- Major division between hardware and software

The "Machine"

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

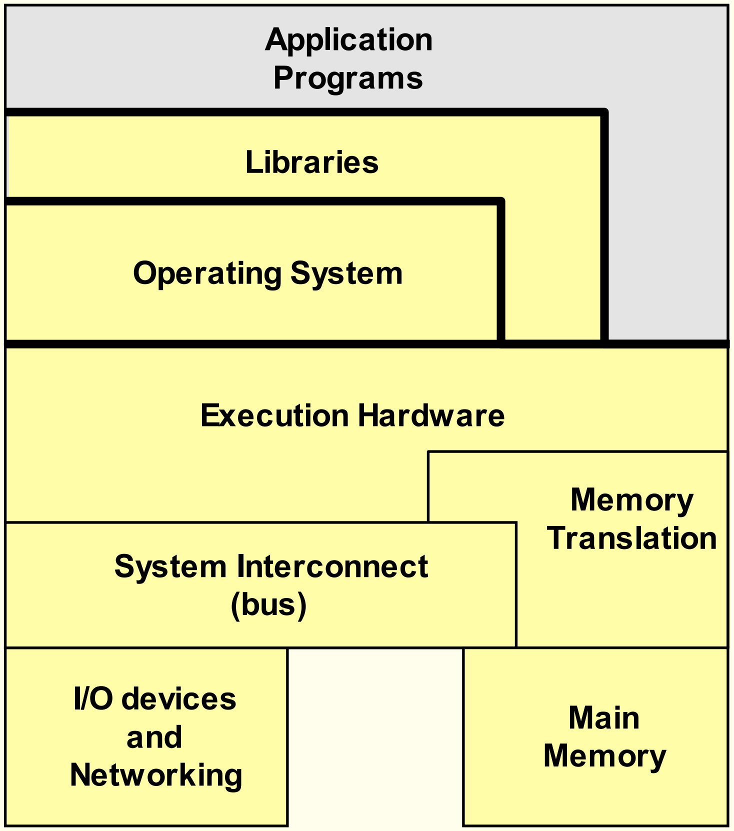


Application Binary Interface

- ABI
- User ISA + OS calls

The “Machine”

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



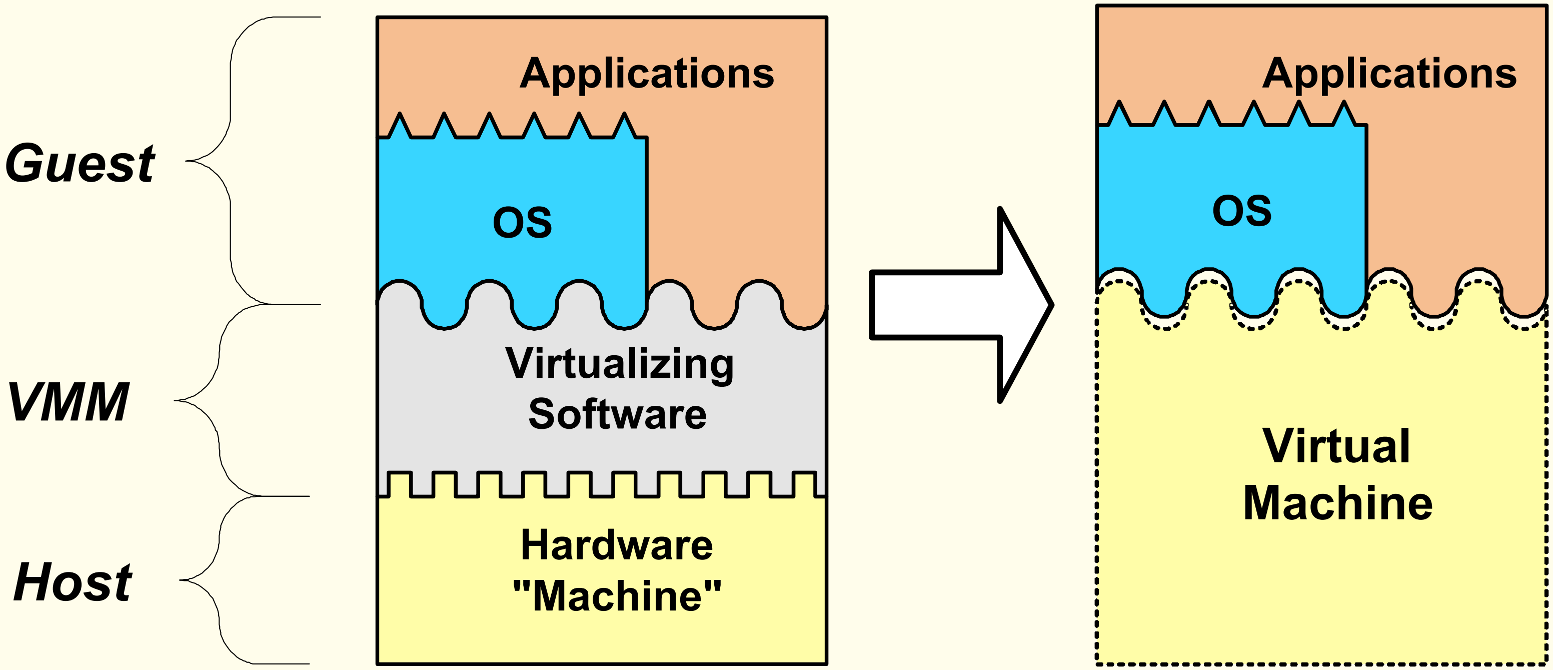
Application Program Interface

- API
- User ISA + library calls

Virtual Machines

add *Virtualizing Software* to a *Host* platform
and support *Guest* process or system on a *Virtual Machine* (VM)

Example: System Virtual Machine



The Family of Virtual Machines

- **Lots of things are called "virtual machines"**
  - IBM VM/370
  - Java
  - VMware

Some things *not* called "virtual machines", *are* virtual machines

- IA-32 EL
- Dynamo
- Transmeta Crusoe

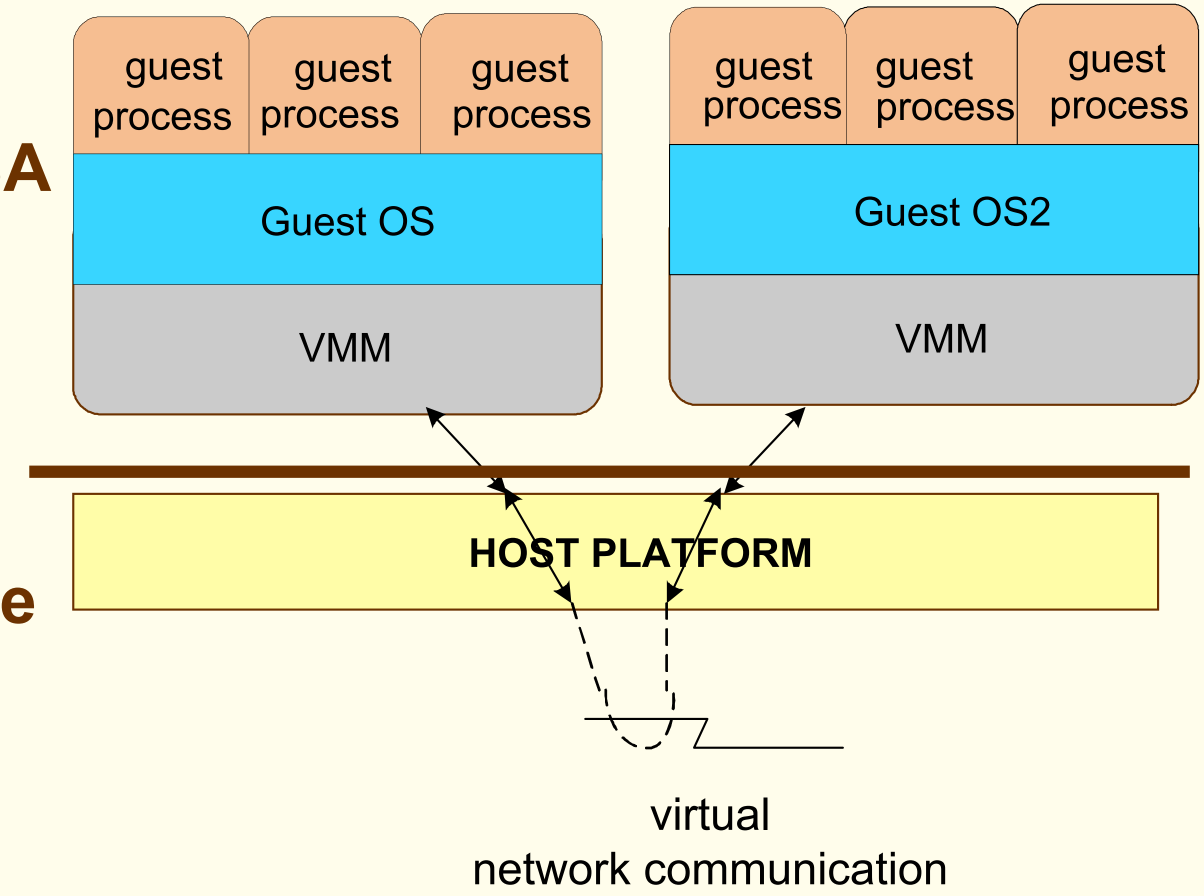
Taking a Unified View

"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

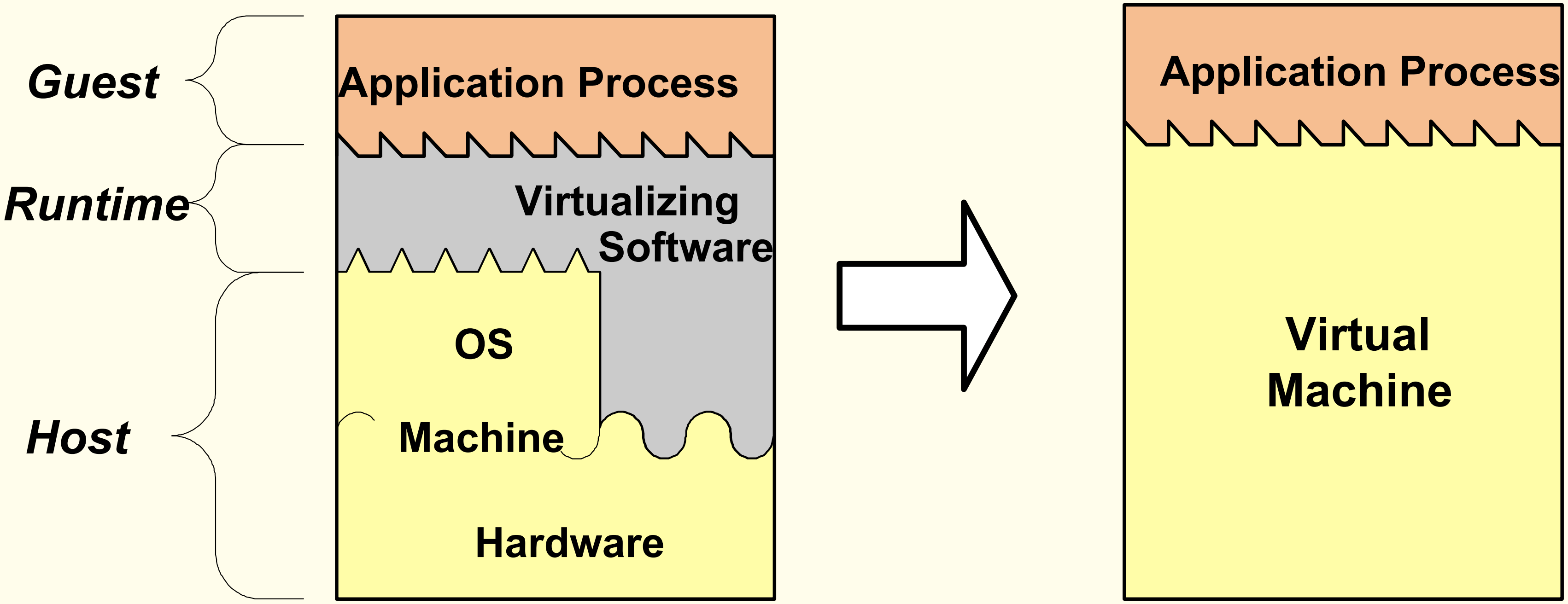
System Virtual Machines

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



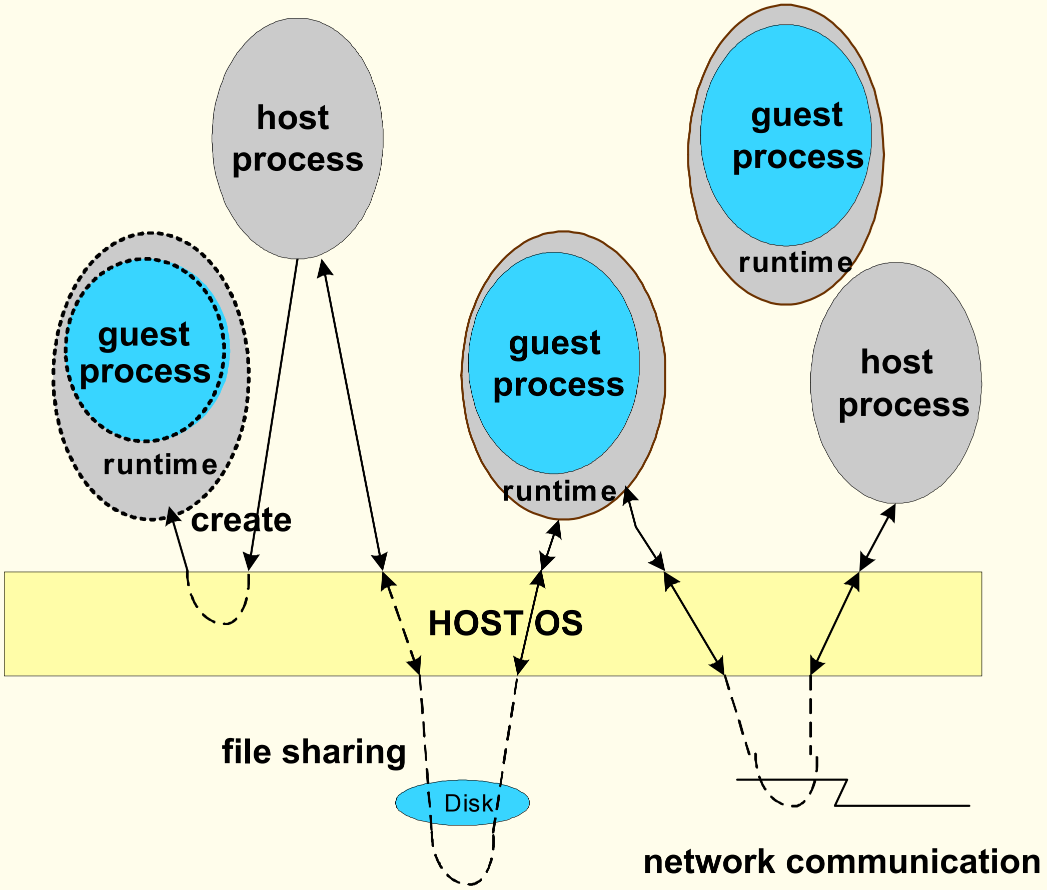
Process VMs

- Execute application binaries with an ISA *different* from hardware platform
- Couple at ABI level via *Runtime System*
- Not persistent



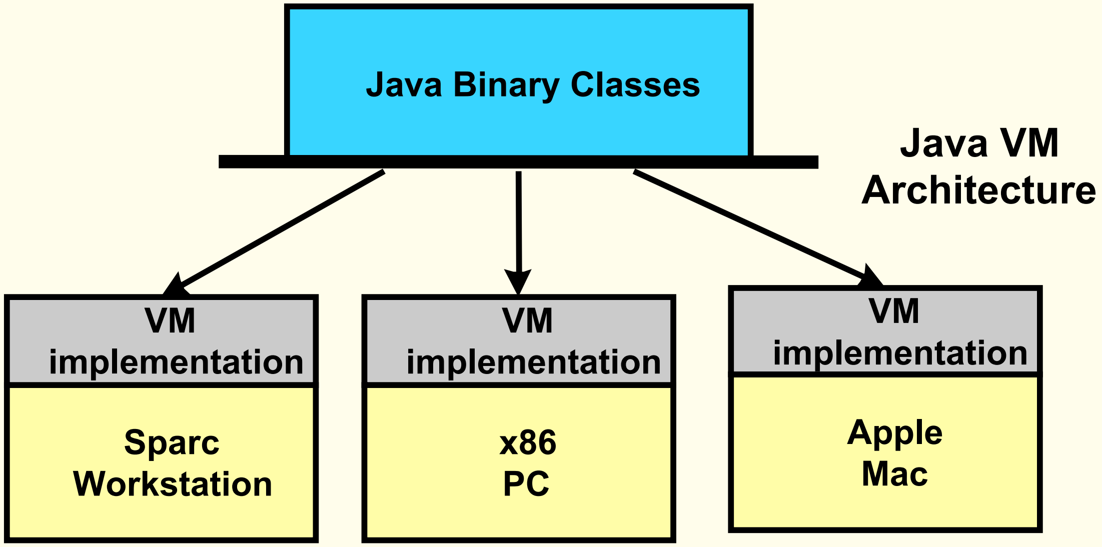
Process Virtual Machines

- Guest processes may intermingle with host processes
- As a practical matter, guest and host OSes are often the same
- Same-ISA Dynamic optimizers are a special case
- Examples: IA-32 EL, FX!32, Dynamo



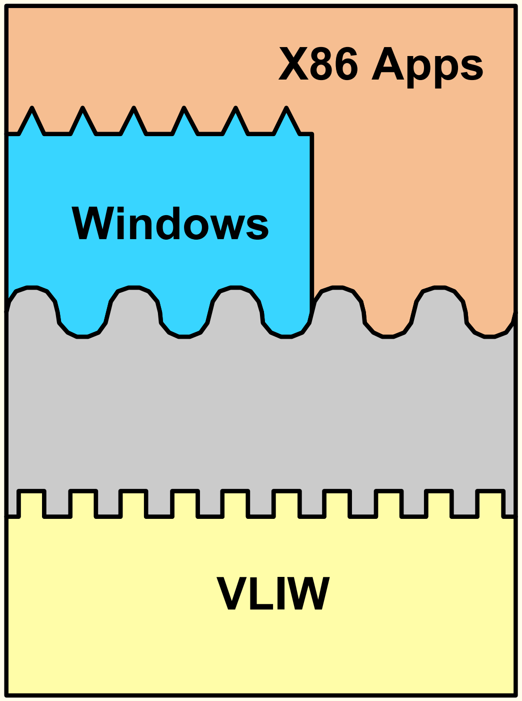
HLL VMs

- Java and CLI are recent examples
- Binary class files are distributed
- “ISA” is part of binary class format
- OS interaction via APIs (part of VM platform)

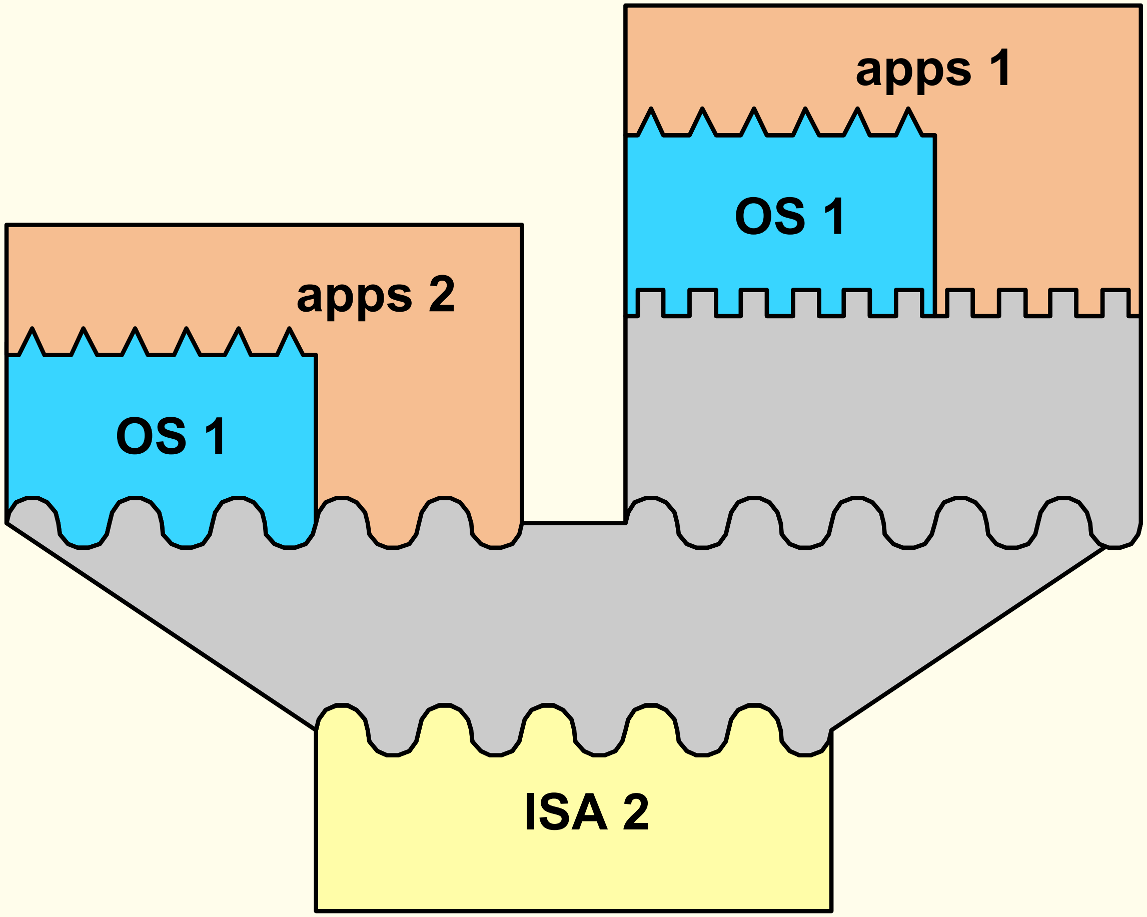


Co-Designed VMs

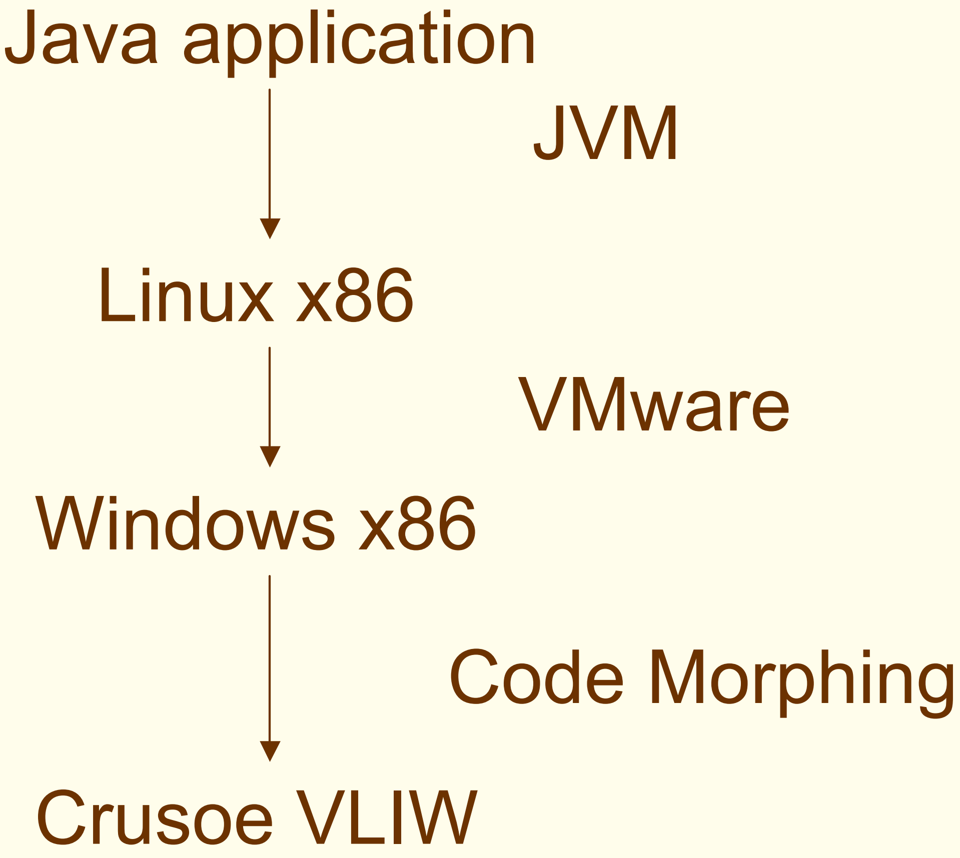
- Perform both translation and optimization
- VM provides interface between standard ISA software and implementation ISA
- Primary goal is performance or power efficiency
- Use proprietary implementation ISA
- Transmeta Crusoe and IBM Daisy best-known examples



Composition



Composition: Example



Java application

JVM

Linux x86

VMware

Windows x86

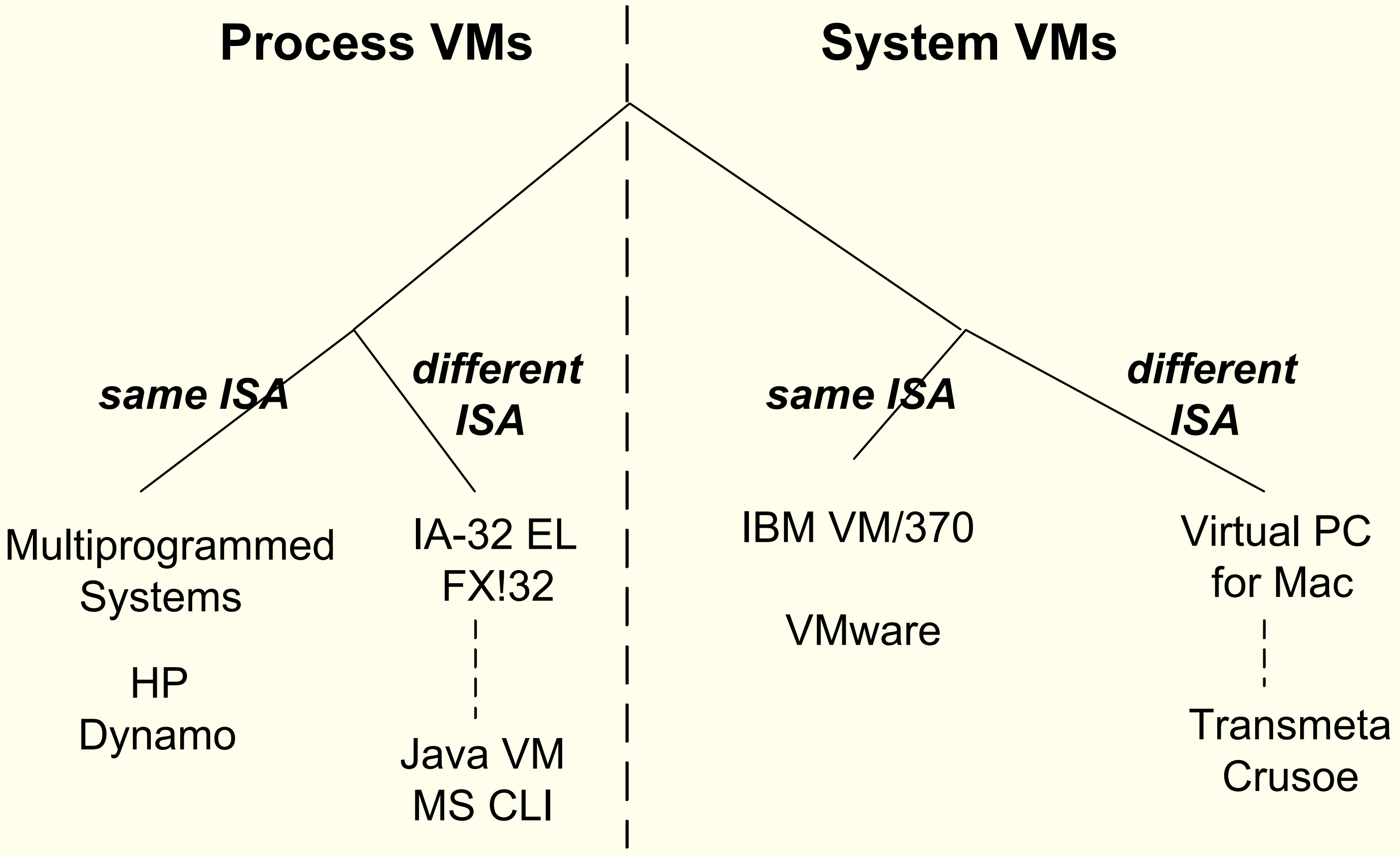
Code Morphing

Crusoe VLIW

Summary (Taxonomy)

VM type (Process or System)

Host/Guest ISA same or different



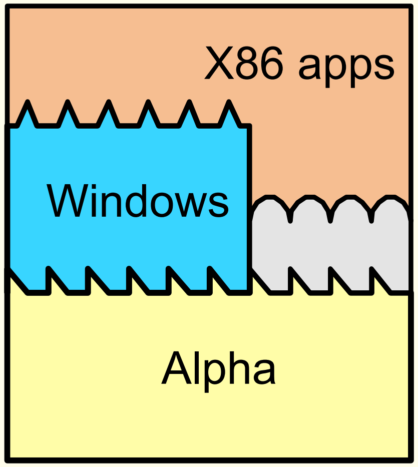
Tutorial Topics

- ❑ Introduction & VM Overview
- ❑ Emulation: Interpretation & Binary Translation
- ❑ Process VMs & Dynamic Translators
- ❑ HLL VMs
- ❑ Co-Designed VMs
- ❑ System VMs

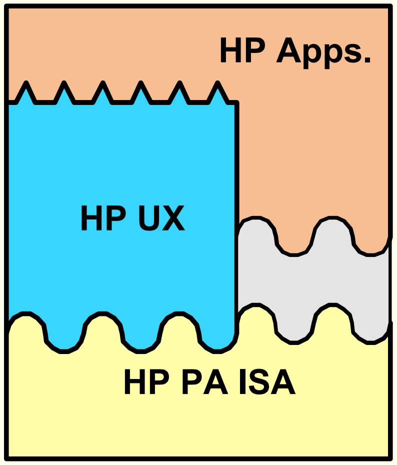
Emulation: Interpretation and Binary Translation

Key VM Technologies

- *Emulation:* binary in one ISA is executed on processor supporting a *different* ISA
- *Dynamic Optimization:* binary is improved for higher performance
  - May be done as part of emulation
  - May optimize *same* ISA (no emulation needed)



Emulation



Optimization

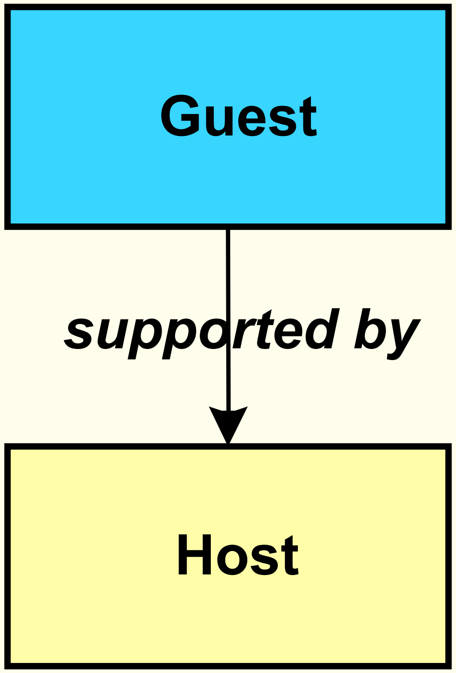
Definitions

NOTE -- there are no standard definitions…

- ***Emulation*:**
  - A method for enabling a (sub)system to present the same interface and characteristics as another.
  - E.g. the execution of programs compiled for instruction set A on a machine that executes instruction set B.
- **Ways of implementing emulation**
  - *Interpretation:* relatively inefficient instruction-at-a-time
  - *Binary Translation:* block-at-a-time optimized for repeated instruction executions

Definitions

- ***Guest***
  - Environment that is being supported by underlying platform
- ***Host***
  - Underlying platform that provides guest environment

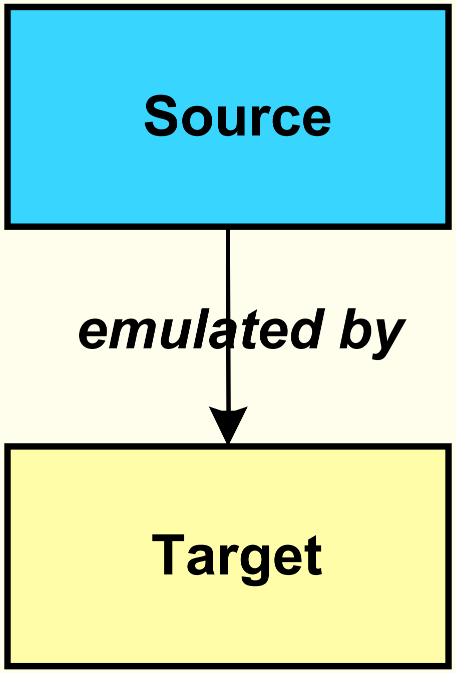


Definitions

- ***Source ISA or binary***
  - Original instruction set or binary
    I.e. the instruction set to be emulated
- ***Target ISA or binary***
  - Instruction set being executed by processor performing emulation
    I.e. the underlying instruction set
    Or the binary that is actually executed

Sometimes Confusing terminology, e.g. shade:

Target -> Host



- **Source/Target refer to ISAs; Guest/Host refer to platforms.**

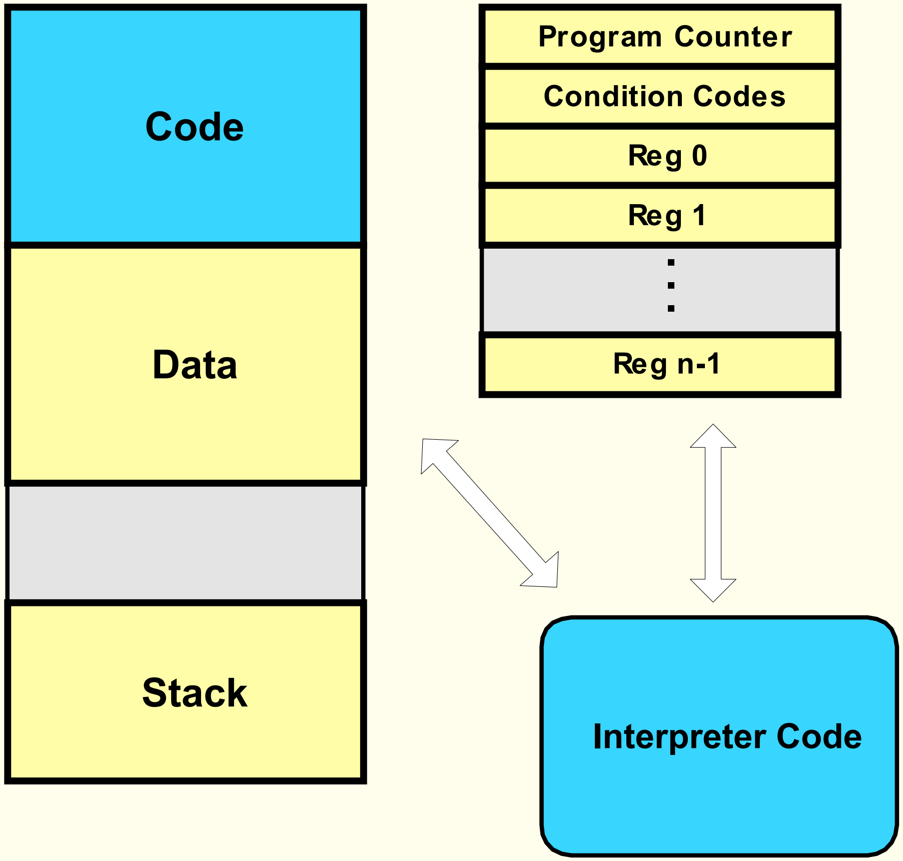
Interpreters

- **HLL Interpreters have a very long history**
  - Lisp
  - Perl
  - Forth (notable for its threaded interpretation model)
- **Binary interpreters use many of the same techniques**
  - Often simplified
  - Some performance tradeoffs are different

    E.g. the significance of using an intermediate form

Interpreter State

- Hold complete source state in interpreter's data memory



Decode-Dispatch Interpretation

```
while (!halt) {
   inst = code(PC);
   opcode = extract(inst,31,6);
   switch(opcode) {
       case LdWordAndZero:LdWordAndZero(inst);
       case ALU: ALU(inst);
       case Branch: Branch(inst);
       . . .}
}
```

Instruction function list

Instruction Functions: Load

```
LdWordAndZero(inst){
     RT = extract(inst,25,5);
     RA = extract(inst,20,5);
     displacement =
extract(inst,15,16);
     source = regs[RA];
     address = source + displacement ;
     regs[RT] = data[address];
     PC = PC + 4;
}
```

Decode-Dispatch: Efficiency

- ❑ **Decode-Dispatch Loop**
  - Mostly serial code
  - Several jumps/branches (some hard-to-predict)
- ❑ **Executing an add instruction**
  - Approximately 20 target instructions
  - Several loads/stores
  - Several shift/mask steps
- ❑ **Hand-coding can lead to better performance**
  - Example: DEC/Compaq FX!32

    Software pipelined decode-dispatch loop

Binary Translation

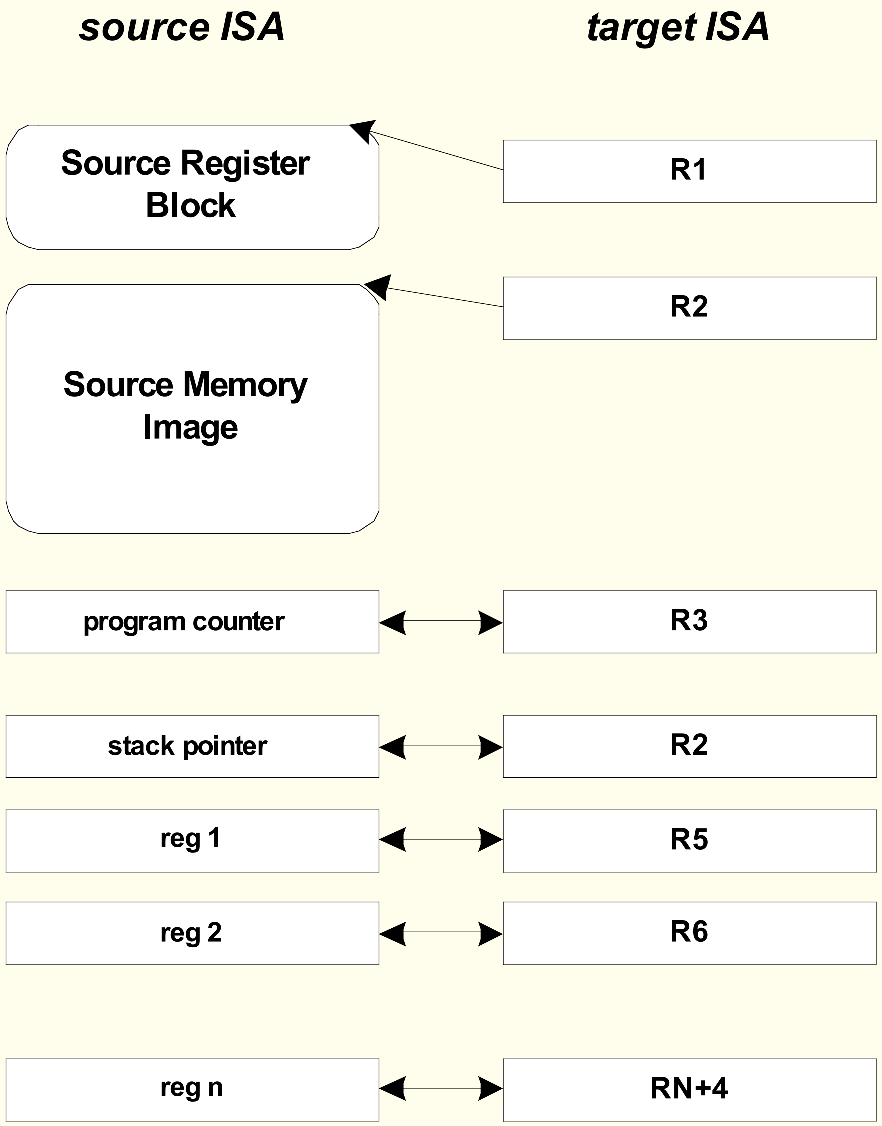
- Generate “custom” code for every source instruction
  - Get rid of repeated instruction “parsing” and jumps altogether

Optimization: Register Mapping

- ❑ Reduces loads/stores significantly
- ❑ Easier if

  #target regs > #source regs
- ❑ Register mapping may be on a per-block basis

  If #target registers not enough



source ISA	target ISA
Source Register Block	R1
Source Memory Image	R2
program counter	R3
stack pointer	R2
reg 1	R5
reg 2	R6
reg n	RN+4

Binary Translation Example

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
r4 holds x86 register %eax
r7 holds x86 register %edx
         etc.
```

Binary Translation Example

x86 Source Binary

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

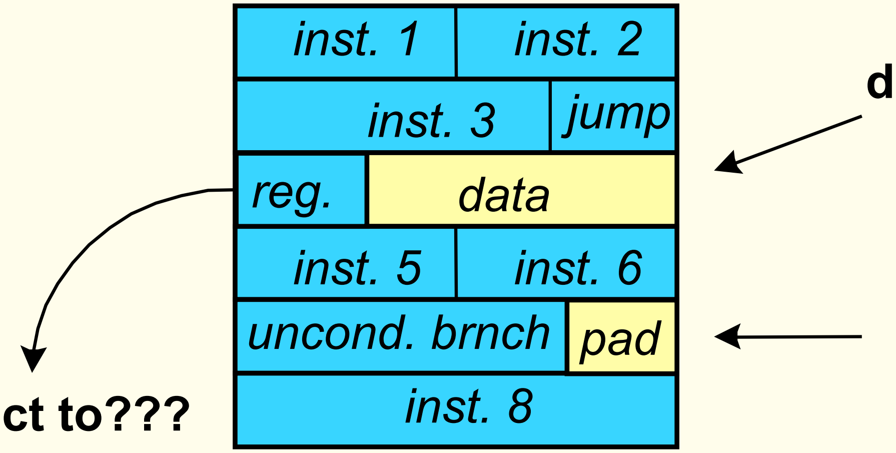
PowerPC Target

```
addi      r16,r4,4     ;add 4 to %eax
lwzx      r17,r2,r16   ;load operand from memory
add       r7,r17,r7    ;perform add of %edx
stwx      r7,r2,r16    ;store %edx value into memory
mr        r4,r16       ;move update value into %eax
addi      r3,r3,9      ;update PC (9 bytes)
```

The Code Discovery Problem

- In order to translate, emulator must be able to "discover" code
  - Easier said than done; especially w/ x86

source ISA instructions



data in instruction stream

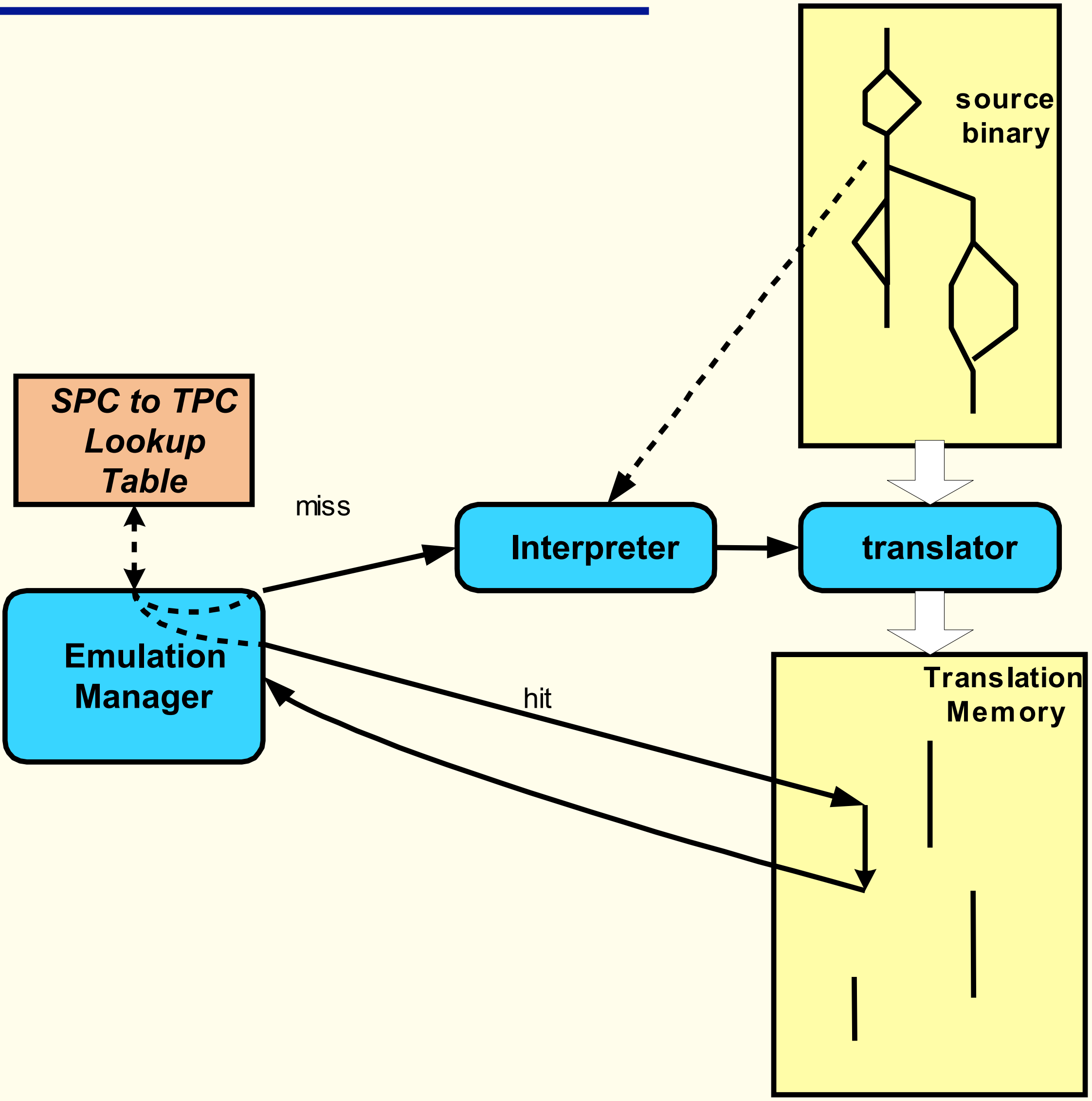
pad for instruction alignment

jump indirect to???

Dynamic Translation

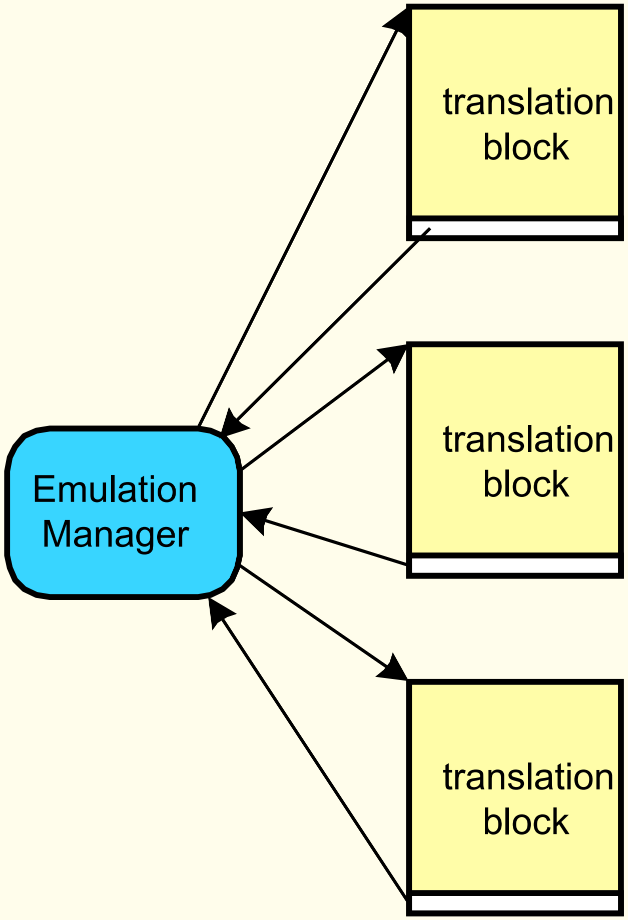
- ❑ **First Interpret**
  - And perform code discovery as a byproduct
- ❑ **Translate Code**
  - Incrementally, as it is discovered
  - Place translated blocks into Code Cache
  - Save source to target PC mappings in lookup table
- ❑ **Emulation process**
  - Execute translated block to end
  - Lookup next source PC in table
    - If translated, jump to target PC
    - Else interpret and translate

Dynamic Translation



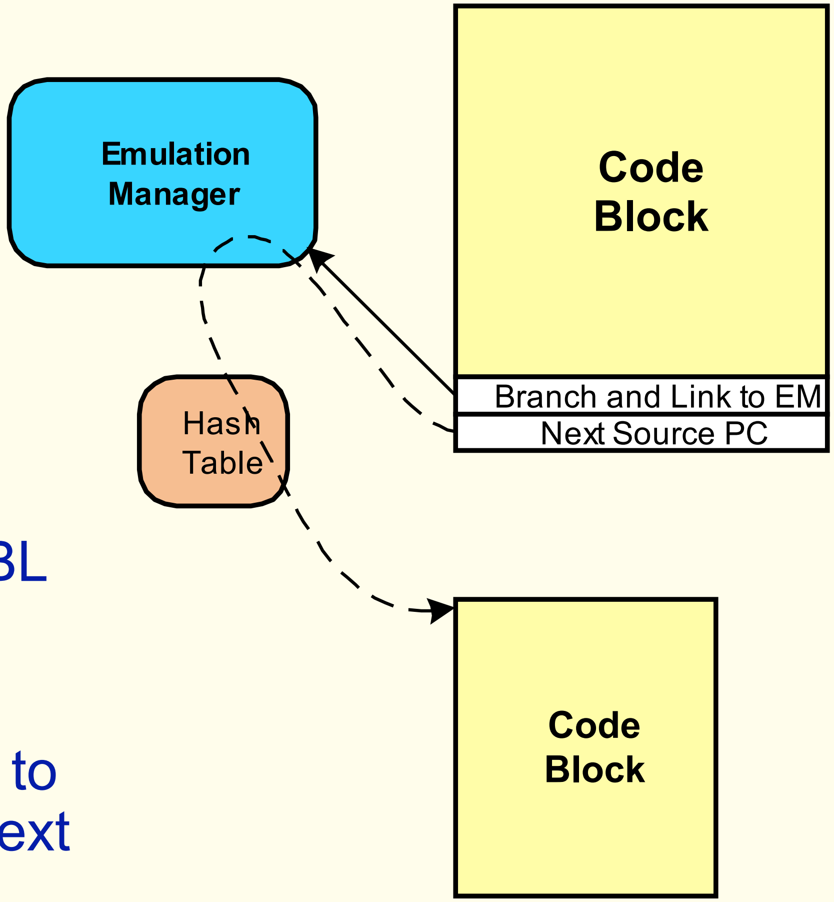
Flow of Control

- Control flows between translated blocks and Emulation Manager



Tracking the Source PC

- Can always update SPC as part of translated code
- Better to place SPC in *stub*



General Method:

- Translator returns to EM via BL
- Source PC placed in stub immediately after BL
- EM can then use link register to find source PC and hash to next target code block

Example

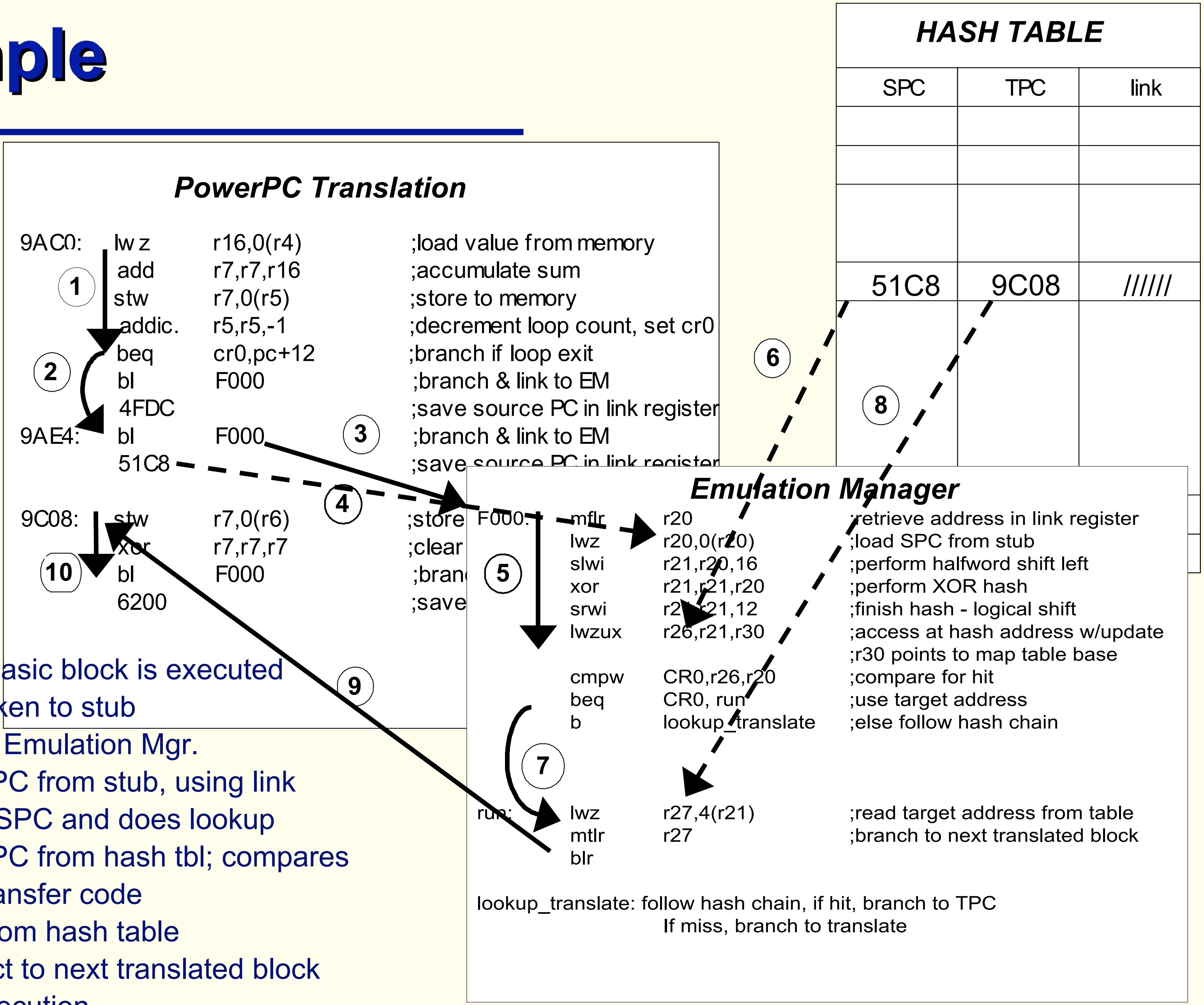
x86 Binary

```
4FD0:   addl     %edx,(%eax)     ;load and accumulate sum
        movl     (%eax),%edx      ;store to memory
        sub      %ebx,1          ;decrement loop count
        jz       51C8            ;branch if at loop end
4FDC:   add      %eax,4           ;increment %eax
        jmp      4FD0             ;jump to loop top


51C8:   movl     (%ecx),%edx      ;store last value of %edx
        xorl     %edx,%edx         ;clear %edx
        jmp      6200            ;jump elsewhere
```

PowerPC Translation

```
9AC0:   lwz      r16,0(r4)          ;load value from memory
        add      r7,r7,r16          ;accumulate sum
        stw      r7,0(r5)           ;store to memory
        addic.   r5,r5,-1           ;decrement loop count, set cr0
        beq      cr0,pc+12          ;branch if loop exit
        bl       F000               ;branch & link to EM
        4FDC                        ;save source PC in link register
9AE4:   bl       F000               ;branch & link to EM
        51C8                        ;save source PC in link register

9C08:   stw      r7,0(r6)           ;store last value of %edx
        xor      r7,r7,r7           ;clear %edx
        bl       F000               ;branch & link to EM
        6200                        ;save source PC in link register
```

Example



HASH TABLE

SPC	TPC	link
51C8	9C08	//////

PowerPC Translation

```
9AC0:   lwz     r16,0(r4)       ;load value from memory
        add     r7,r7,r16       ;accumulate sum
        stw     r7,0(r5)        ;store to memory
        addic.  r5,r5,-1        ;decrement loop count, set cr0
        beq     cr0,pc+12       ;branch if loop exit
        bl      F000            ;branch & link to EM
        4FDC                    ;save source PC in link register
9AE4:   bl      F000            ;branch & link to EM
        51C8                    ;save source PC in link register

9C08:   stw     r7,0(r6)        ;store
        xor     r7,r7,r7        ;clear
        bl      F000            ;bran
        6200                    ;save
```

Emulation Manager

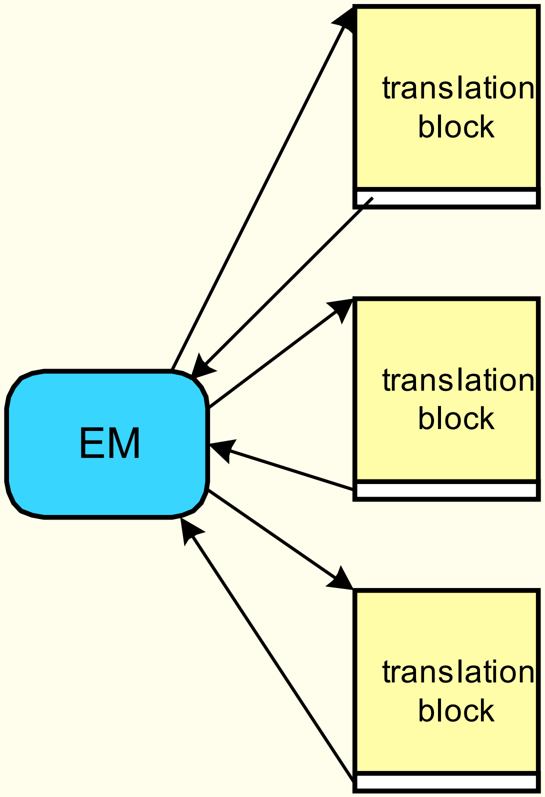
```
F000:   mflr    r20                 ;retrieve address in link register
        lwz     r20,0(r20)          ;load SPC from stub
        slwi    r21,r20,16          ;perform halfword shift left
        xor     r21,r21,r20         ;perform XOR hash
        srwi    r21,r21,12          ;finish hash - logical shift
        lwzux   r26,r21,r30         ;access at hash address w/update
                                    ;r30 points to map table base
        cmpw    CR0,r26,r20         ;compare for hit
        beq     CR0, run            ;use target address
        b       lookup_translate    ;else follow hash chain


run:    lwz     r27,4(r21)          ;read target address from table
        mtlr    r27                 ;branch to next translated block
        blr

lookup_translate: follow hash chain, if hit, branch to TPC
                  If miss, branch to translate
```

1 Translated basic block is executed
2 Branch is taken to stub
3 Stub BAL to Emulation Mgr.
4 EM loads SPC from stub, using link
5 EM hashes SPC and does lookup
6 EM loads SPC from hash tbl; compares
7 Branch to transfer code
8 Load TPC from hash table
9 Jump indirect to next translated block
10 Continue execution

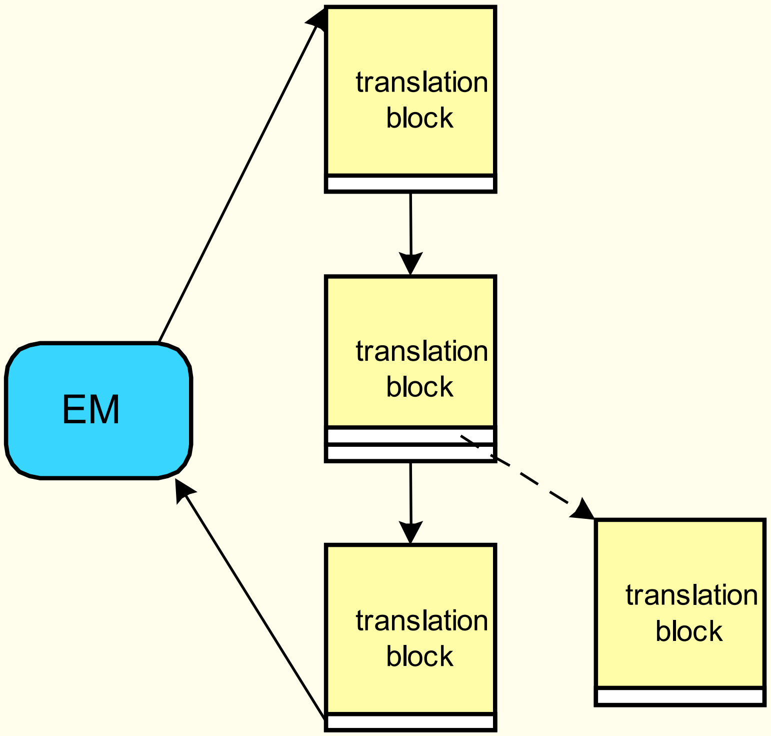
Translation Chaining

- ❑ **Jump from one translation directly to next**
  - Avoid switching back to Emulation Mgr.

Without Chaining:



With Chaining:



Software Jump Prediction

- Form of “Inline Caching”
- Example Code:

  Say Rx holds source branch address

  - `addr_i` are predicted addresses (in probability order)

    Determined via profiling
  - `target_i` are corresponding target code blocks

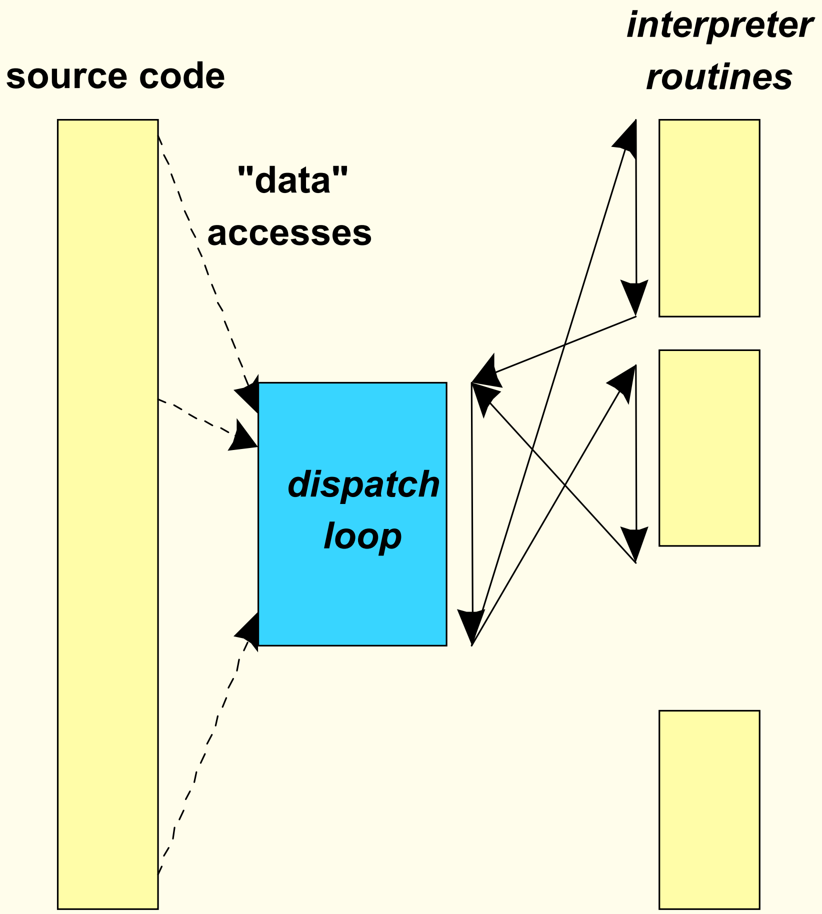
```
If Rx == addr_1 goto target_1
Else if Rx == addr_2 goto target_2
Else if Rx == addr_3 goto target_3
Else hash_lookup(Rx) ; do it the slow way
```

Source/Target ISA Issues

- **Register architectures**
- **Condition codes**
  - Lazy evaluation as needed
- **Data formats and operations**
  - Floating point
  - Decimal
  - MMX
- **Address resolution**
  - Byte vs Word addressing
- **Address alignment**
  - Natural vs arbitrary
- **Byte order**
  - Big/Little endian

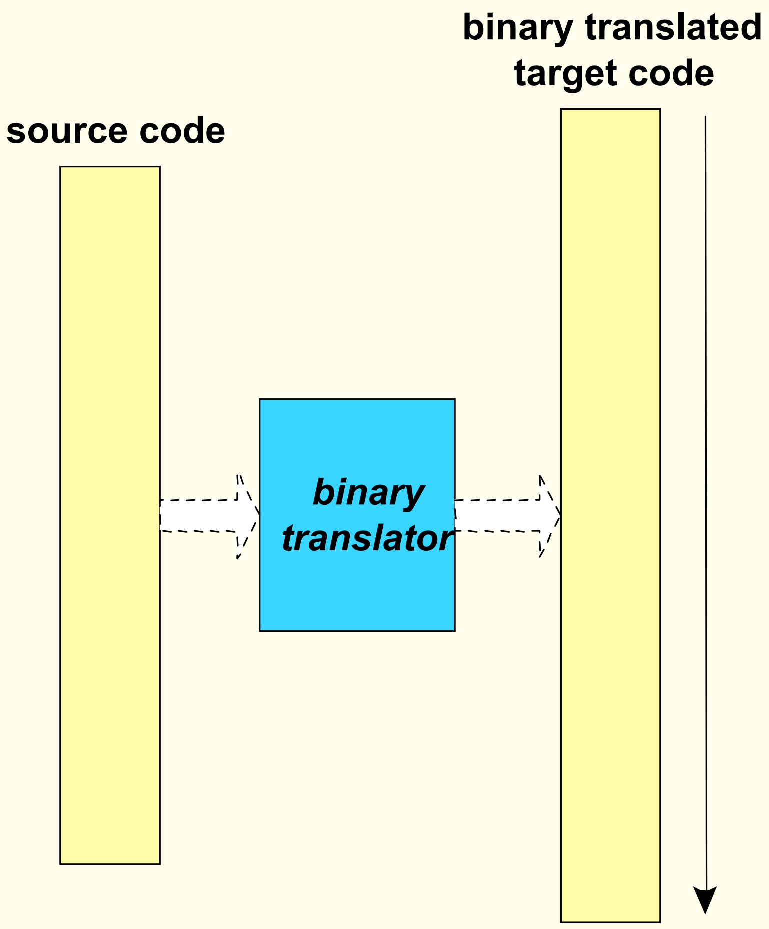
Emulation Summary

- **Decode/Dispatch Interpretation**
  - Memory Requirements: Low
  - Startup: Fast
  - Steady State Performance: Slow
  - Portability: Excellent



Emulation Summary

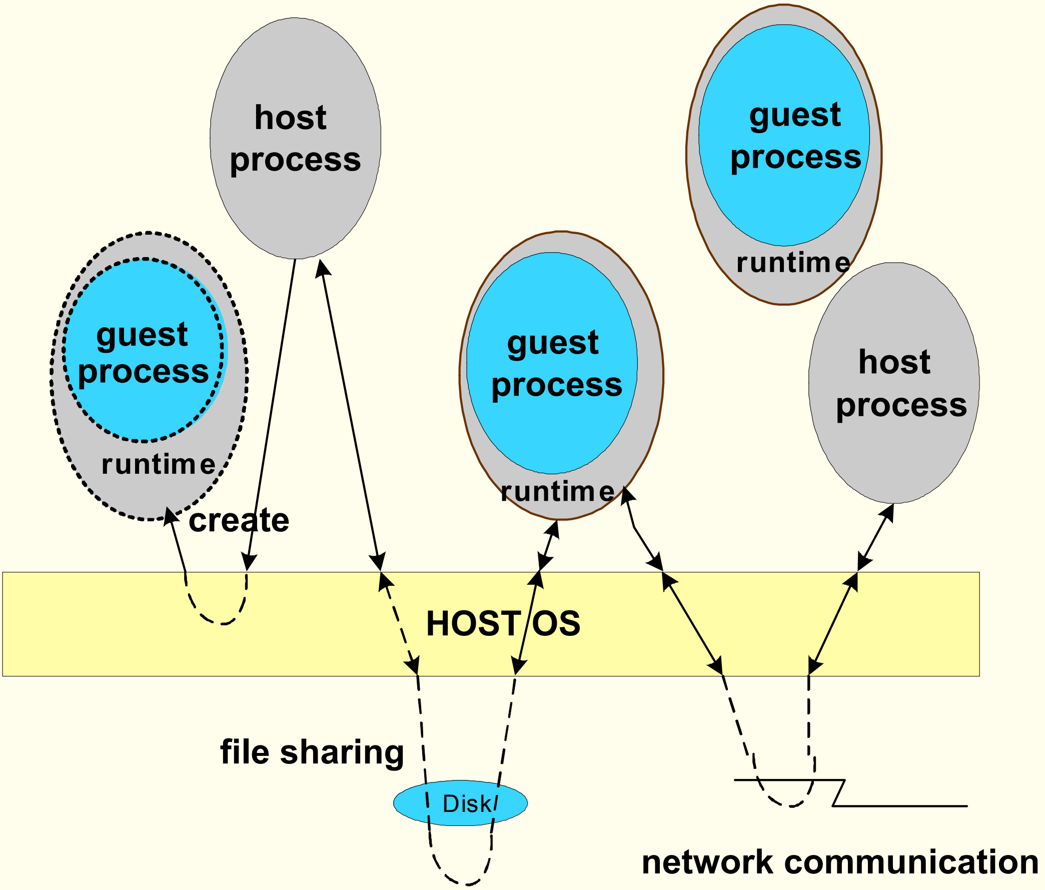
- ❑ **Binary Translation**
  - Memory Requirements: High
  - Startup: Very Slow
  - Steady State Performance Fast
  - Portability: Poor



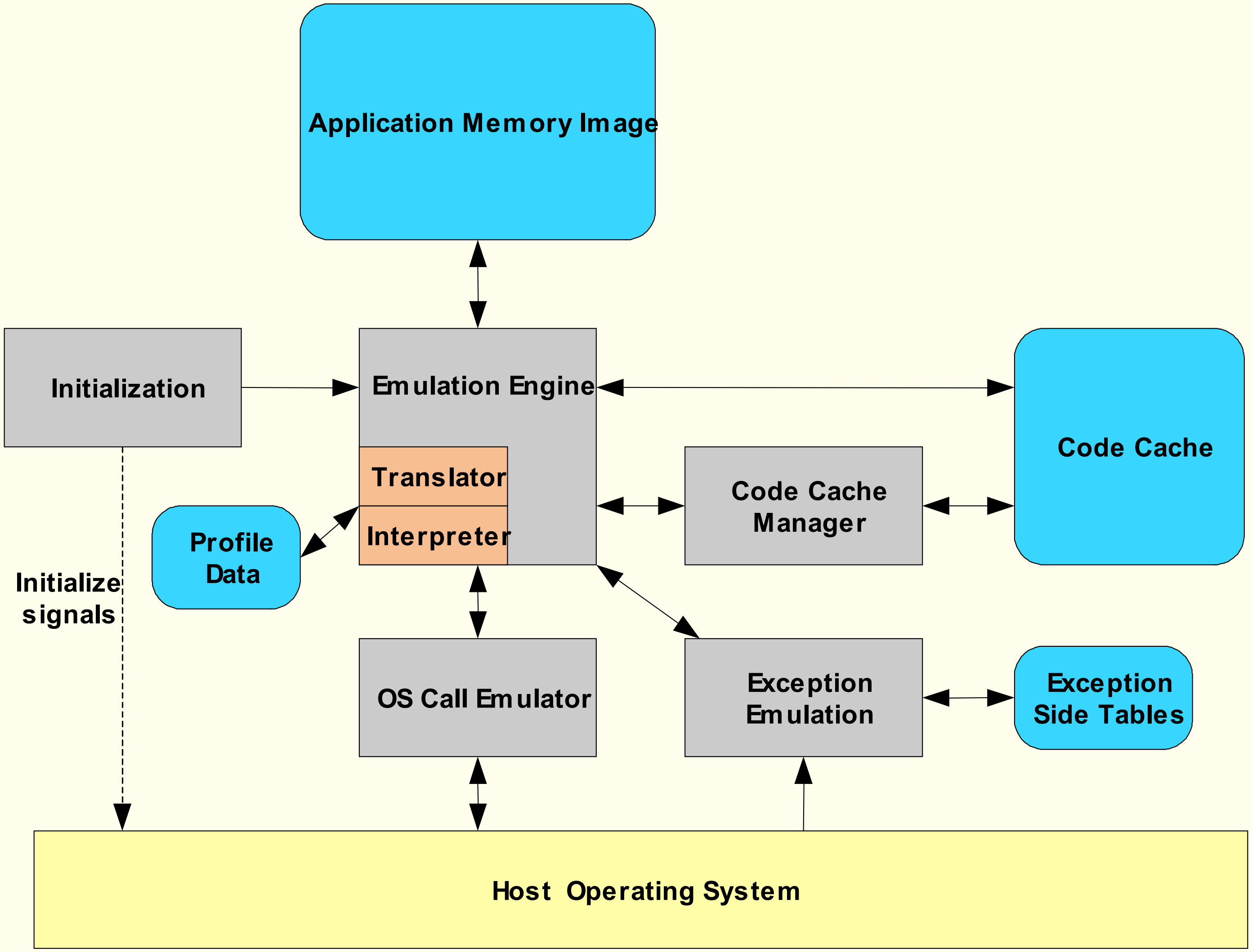
Process Virtual Machines

Process Virtual Machines

- Perform guest/host mapping at ABI level
- Encapsulate guest process in process-level runtime
- Issues
  - Memory Architecture
  - Exception Architecture
  - OS Call Emulation
  - Overall VM Architecture
  - High Performance Implementations
  - System Environments



Process VM Architecture

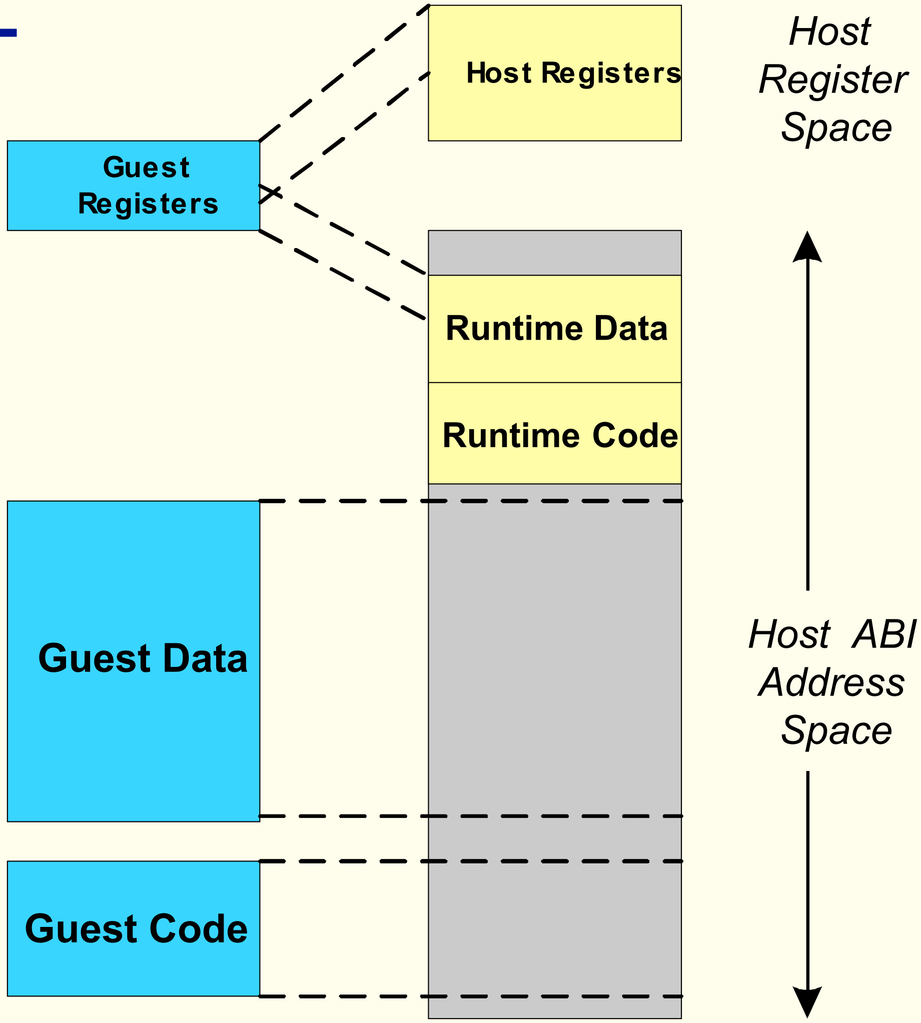


Runtime Components

- ❑ **Initialization**
  - Allocate Memory
  - Initialize runtime data structures
  - Initialize all signals
- ❑ **Code Cache Manager**
  - Implement replacement algorithm when cache fills
  - Flush when required (e.g. self-modifying code)
- ❑ **OS Call Emulator**
  - Translate OS Calls
  - Translate OS Responses
- ❑ **Exception Emulator**
  - Handle signals
    - If registered by source code, pass to emulated source handler
    - If not registered emulate host response
  - Form precise state

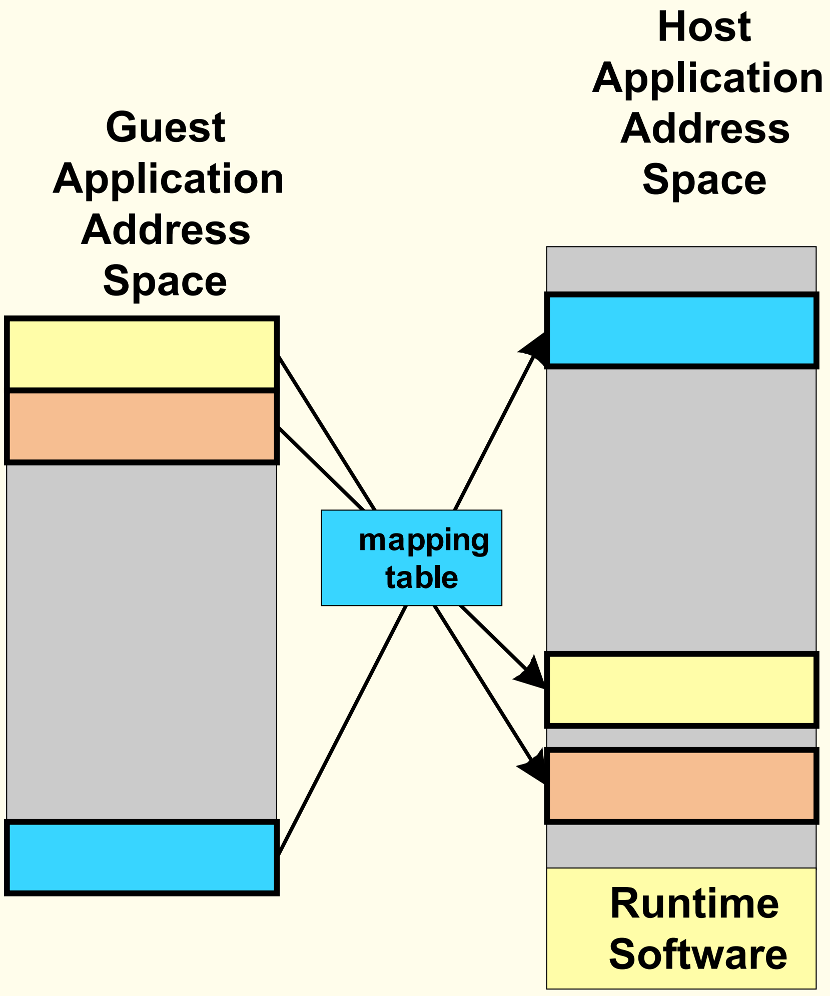
State Mapping

- ❑ **For best performance**
  - Guest register space fits “inside” host register space
  - Guest memory space fits “inside” host memory space
  - Best case does not always happen
  - But often does (x86 on RISC)



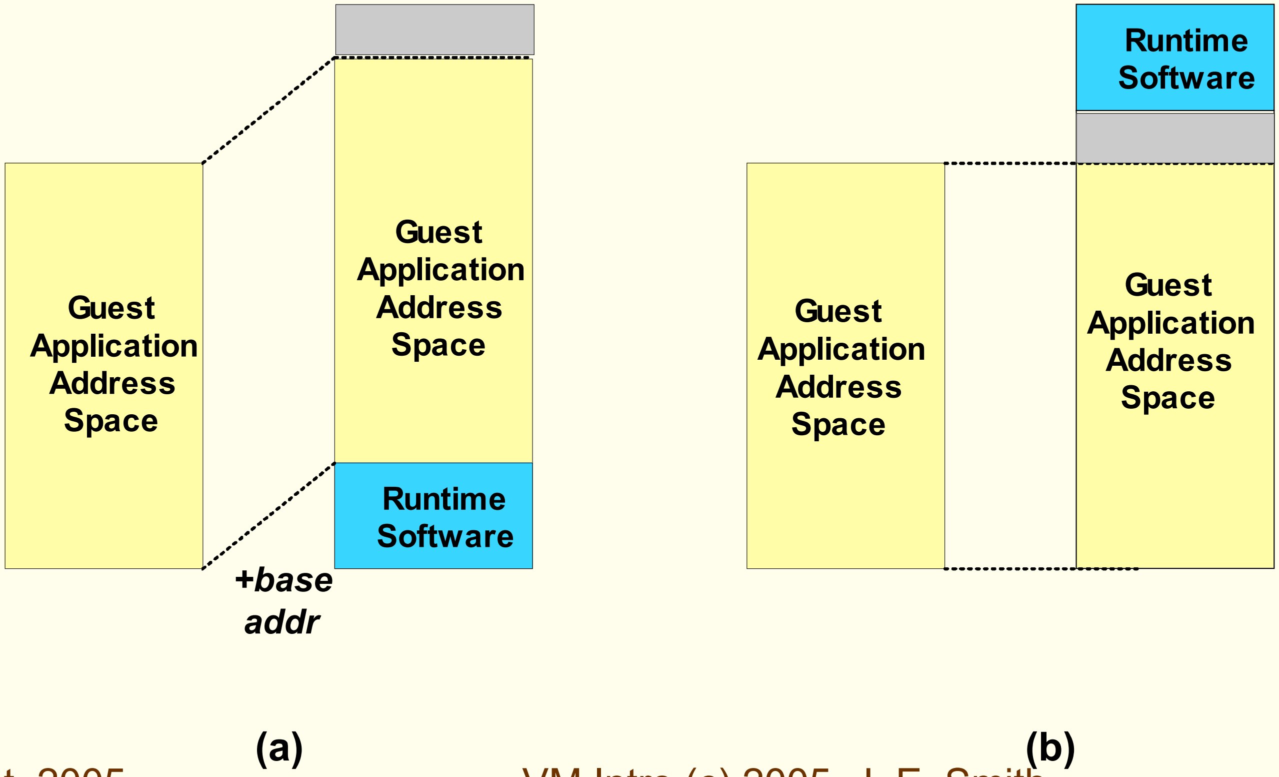
Software Mapping

- ❑ **Runtime Software maintains mapping table**
  - Similar to hardware page tables/TLBs
  - Slow, but can always be made to work


Host Application Address Space
Guest Application Address Space
mapping table
Runtime Software

Direct (Hardware) Mapping

- ❑ **VM software mapping is slow**
  - Several instructions per load/store
- ❑ **Use underlying hardware**
  - If guest address space + runtime fit within host space

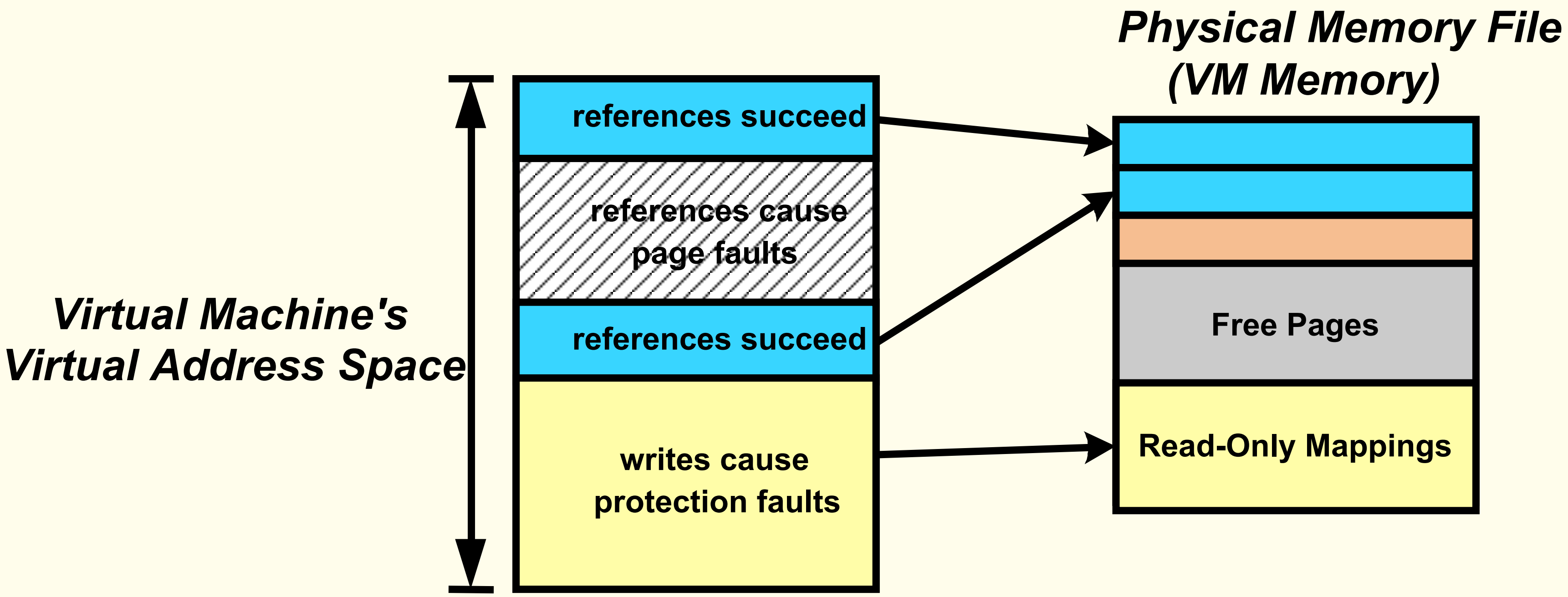

Guest Application Address Space
Guest Application Address Space
Runtime Software
+base addr
(a)
Guest Application Address Space
Runtime Software
Guest Application Address Space
(b)

Guest Memory Protection

- ❑ **Runtime must be protected from Guest process**
- ❑ **VM software mapping can be easily used**
  - Place (and check) protection info in mapping table
- ❑ **Better: use underlying hardware**
  - Runtime must be able to set privileges
  - Protection faults should be reported to Runtime
    (So it can respond as guest OS would – later)
  - Requires some support from Host OS

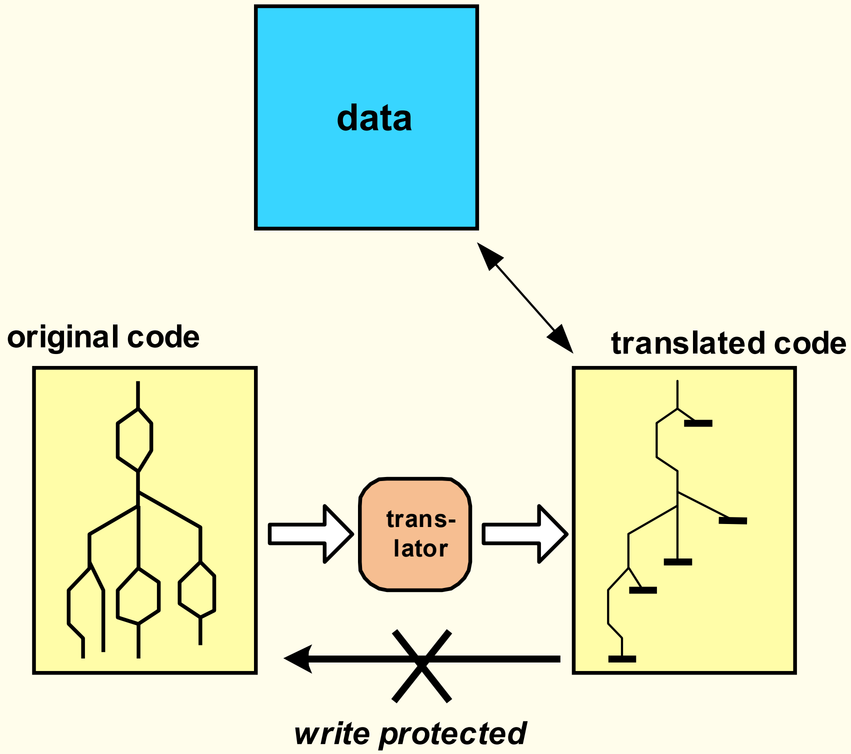
Host OS Support

- A system call where runtime can set protection levels
- A signal mechanism where protection faults trap to handler in runtime
- SimOS Example
  - Map guest space to a file (map, unmap, read-only mapping)
  - Signal (SIGSEGV) delivered to VM software on fault



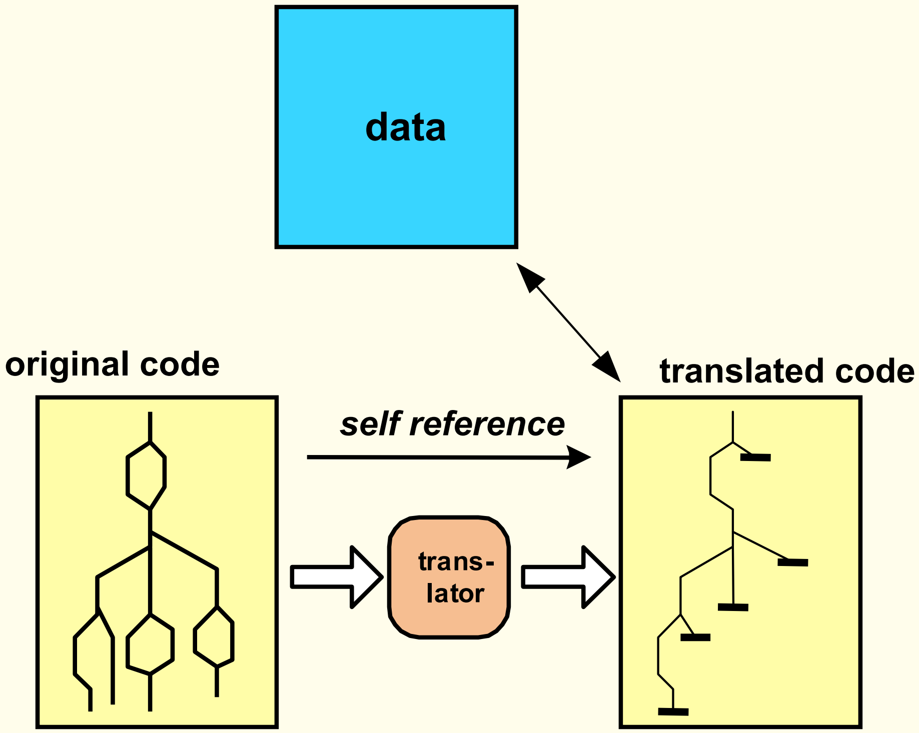
Self-Modifying Code

- ❑ Write-Protect original code to catch code self-modification
- ❑ Exception handler invalidates old translations
- ❑ Be sure to make forward progress
- ❑ Pseudo self-modifying code may require optimizations
  - Data mixed in with code
  - Implement software-supported fine-grain checking (Transmeta)
- ❑ Sometimes can rely on source binary to indicate self modification e.g. SPARC flush



Self-referencing code

- Original copy is maintained by translator
- All reads are with respect to original copy ⇒ correct data is returned



Exceptions: Interrupts

- **Application may register some interrupts**
  - Precise state easier than traps

    (because there is more flexibility wrt location)
- **Problem: Translated blocks may executed for an unbounded time period**
- **Solution:**
  - Interrupt signal goes to runtime
  - Runtime unchains translation block currently executing

    (eliminates loops)
  - Runtime returns control to current translation
  - Translation soon reaches end (and precise state is available)
- **If interrupts are common, runtime may inhibit all chaining**

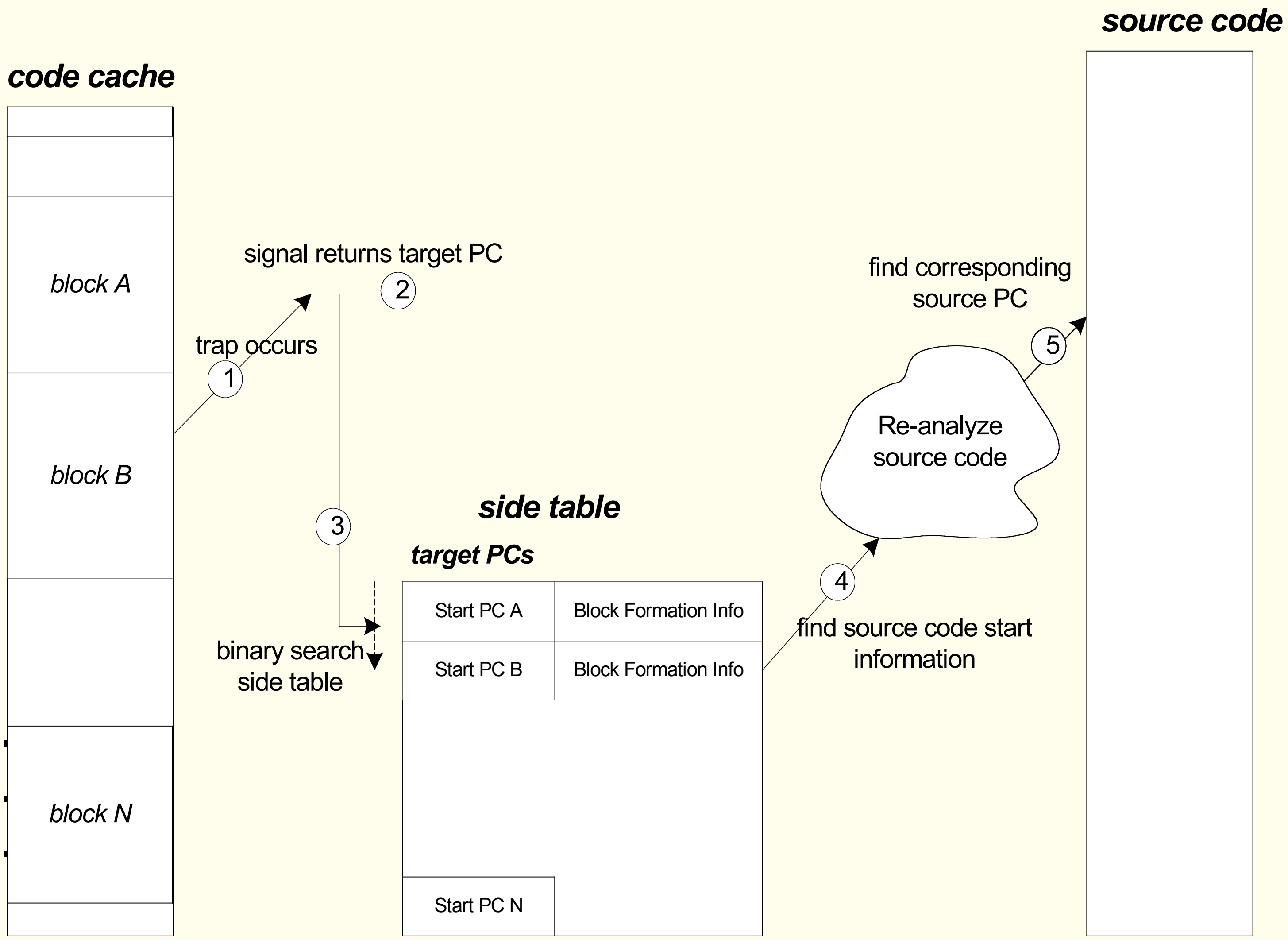
Exceptions: Traps

- **Can be detected directly via interpretation**
  - or explicit translated code checks
- **Can be detected indirectly via target ISA trap and signal**
  - Runtime registers all trap conditions as signals
- **Semantic “matching”**
  - If trap is architecturally similar in target in source then trap/signal may be used
  - Otherwise interpretive method must be used
- **Generally, more difficult than interrupts wrt precise state**

Precise State: Program Counter

- ❑ **Interpretation: Easy – source PC is maintained**
- ❑ **Binary translation: more difficult – source PC only available at translation block boundaries**
  - Trap PC is in terms of target code
  - Target PC must be mapped back to correct source PC
- ❑ **Solution**
  - Use side table and reverse translate
  - Can be combined with PC mapping table
  - Requires search of table to find trapping block
  - Reconstruct block translation to identify specific source PC of trapping instruction

PC Side Table



source code

code cache

block A

block B

block N

1 trap occurs

2 signal returns target PC

3 binary search side table

side table

target PCs

Start PC A	Block Formation Info
Start PC B	Block Formation Info
Start PC N	

4 find source code start information

Re-analyze source code

5 find corresponding source PC

Recovering Register State

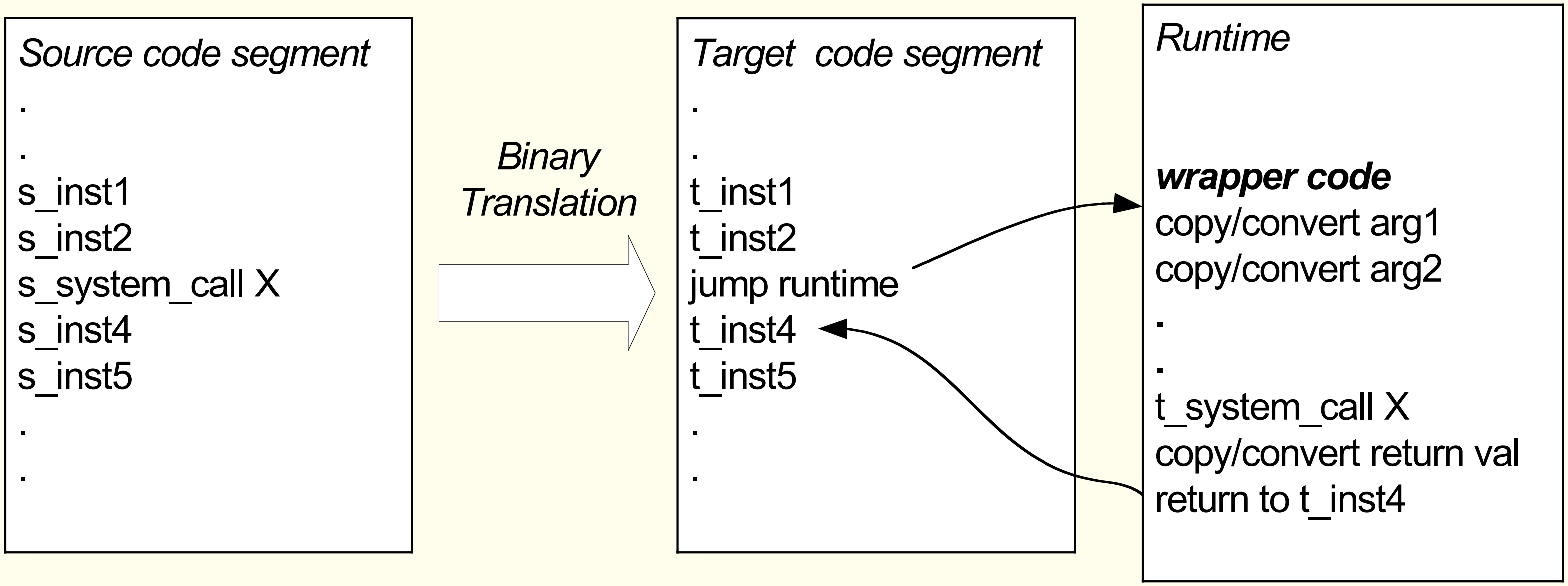
- **Simple if target code updates register state in same order as source code**
  - Register state mapping can be used to generate source register values
- **More difficult if optimizations reorder code**
  - Implement software version of reorder buffer or checkpoints

Recovering Memory State

- **Simple if target code updates memory state in same order as source code**
  - Restricts optimizations (more difficult to back-up than register state)
  - Most process VMs maintain original store order

OS Call Emulation

- "Wrapper" or "Jacket" code converts source call to target OS call(s)



OS Call Emulation

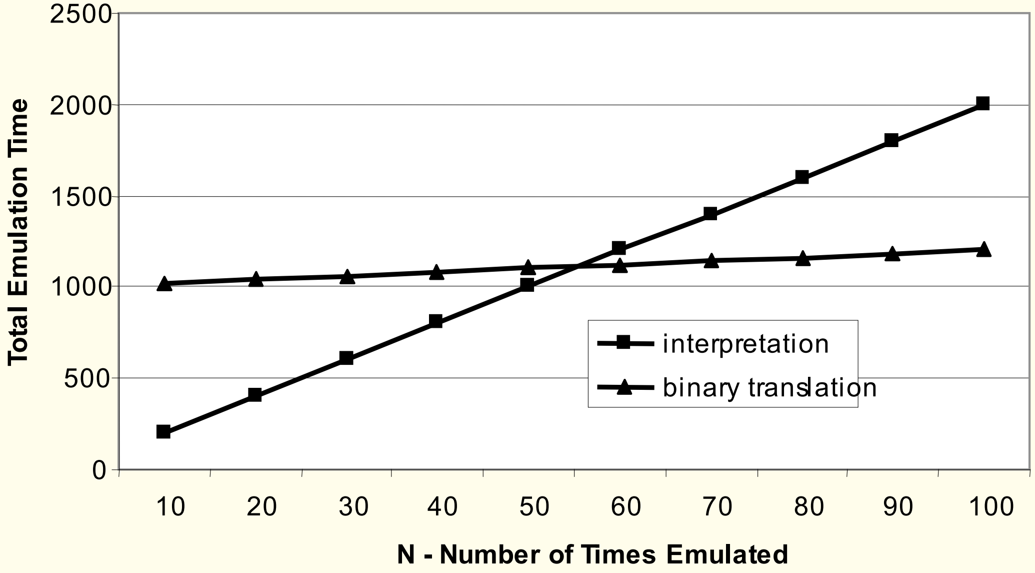
- ❑ **Same source and target OSes (different ISAs)**
  - Syntactic translation only
  - E.g. pass arguments in stack rather than registers
- ❑ **Different source and target OSes**
  - Semantic translation/matching required

    Similar to inter-OS porting
  - May be difficult (or impossible)
  - OS deals with real world

    What if source OS supports a type of device that the target does not?

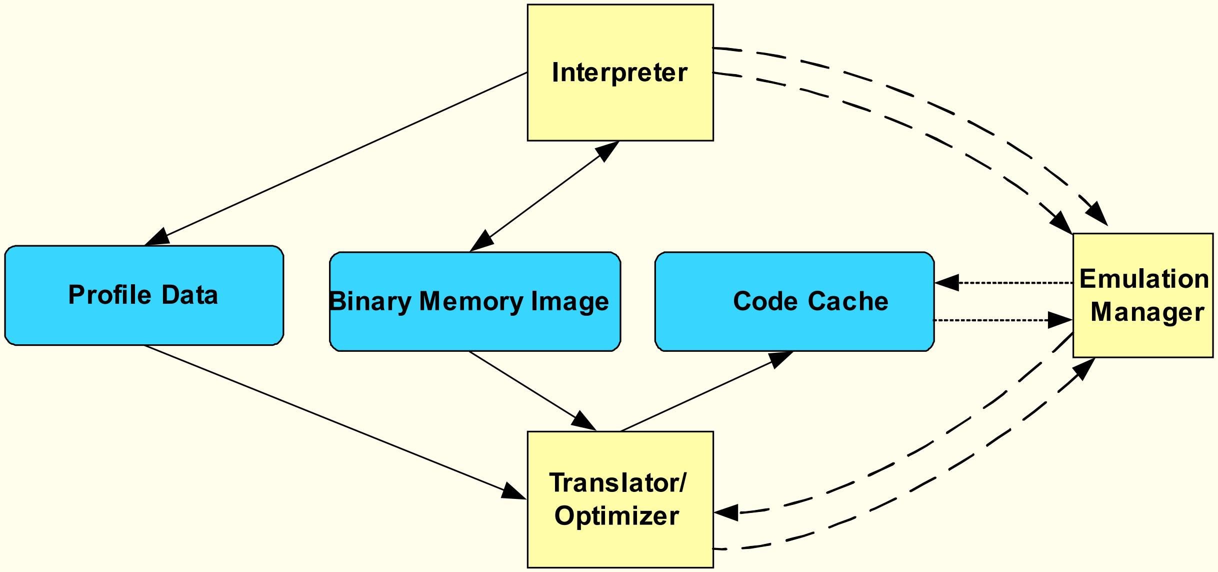
High Performance Emulation

- **Important tradeoff**
  - Startup time -- Cost of converting code for emulation
  - Steady state -- Cost of emulating
- **Interpretation:**
  - Low startup, high steady state cost
- **Binary translation**
  - High startup, low steady state cost



Staged Emulation

- Adjust optimization level to execution frequency
- Tradeoff
  - Total runtime = program runtime + translation overhead
  - Higher optimization ⇒ shorter program runtime
  - Lower optimization ⇒ lower overhead



Staged Emulation

- **General Strategy**
  1. Begin interpreting
  2. For code executed above a threshold

     Use *simple translation/optimization*
  3. For translated code executed above a threshold

     Optimize more
  - etc.
- **Specific Strategies may skip some of the steps**
  - Shade uses 1 and 2
  - Wabi uses 2 and 3
  - FX!32 uses 1 and 3
  - IA32-EL, UQDBT use 2 and 3

Code Cache Management

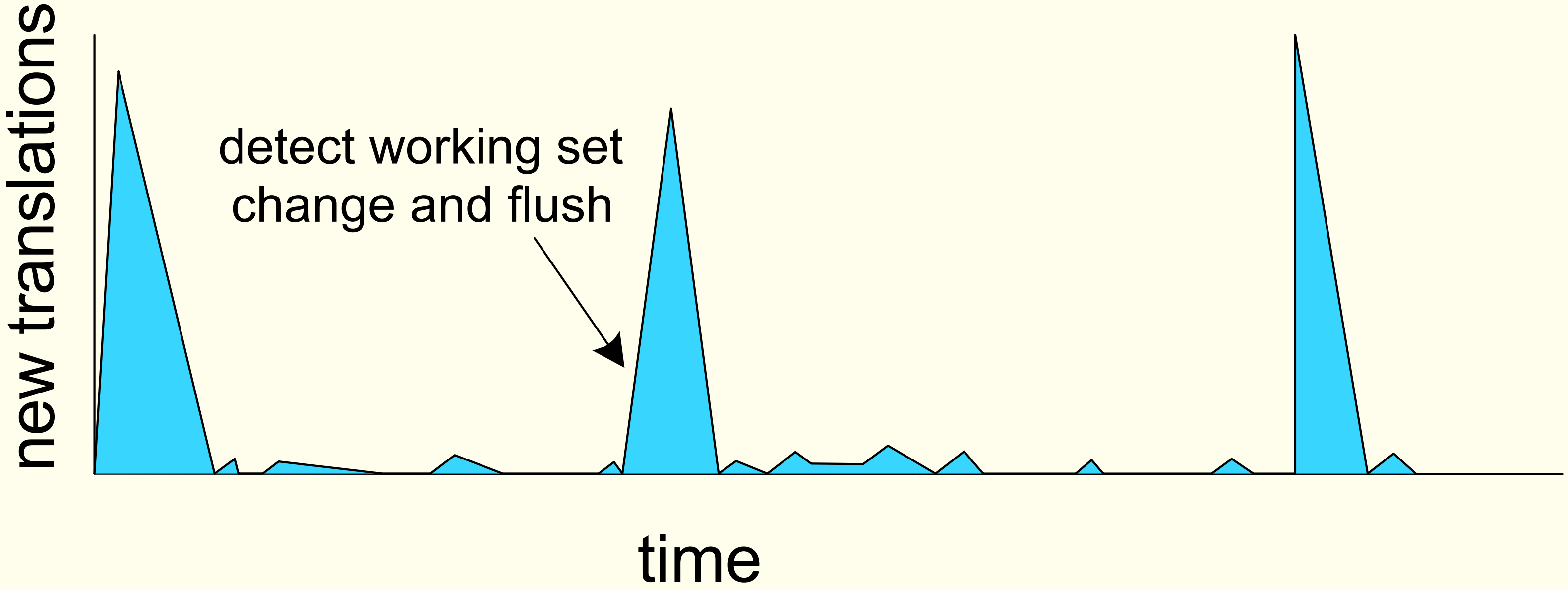
- ❑ **Code Cache is different from typical hardware cache**
  - Variable sized blocks
  - Dependences among blocks due to linking
  - No “backing store”; re-generating is expensive
- ❑ **These factors affect replacement algorithm**
  - LRU replacement is typically not used
    (fragmentation problems)

Flush When Full

- Simple, basic algorithm
- Gets rid of “stale” links if control flow changes
- High overhead for re-translating after flush

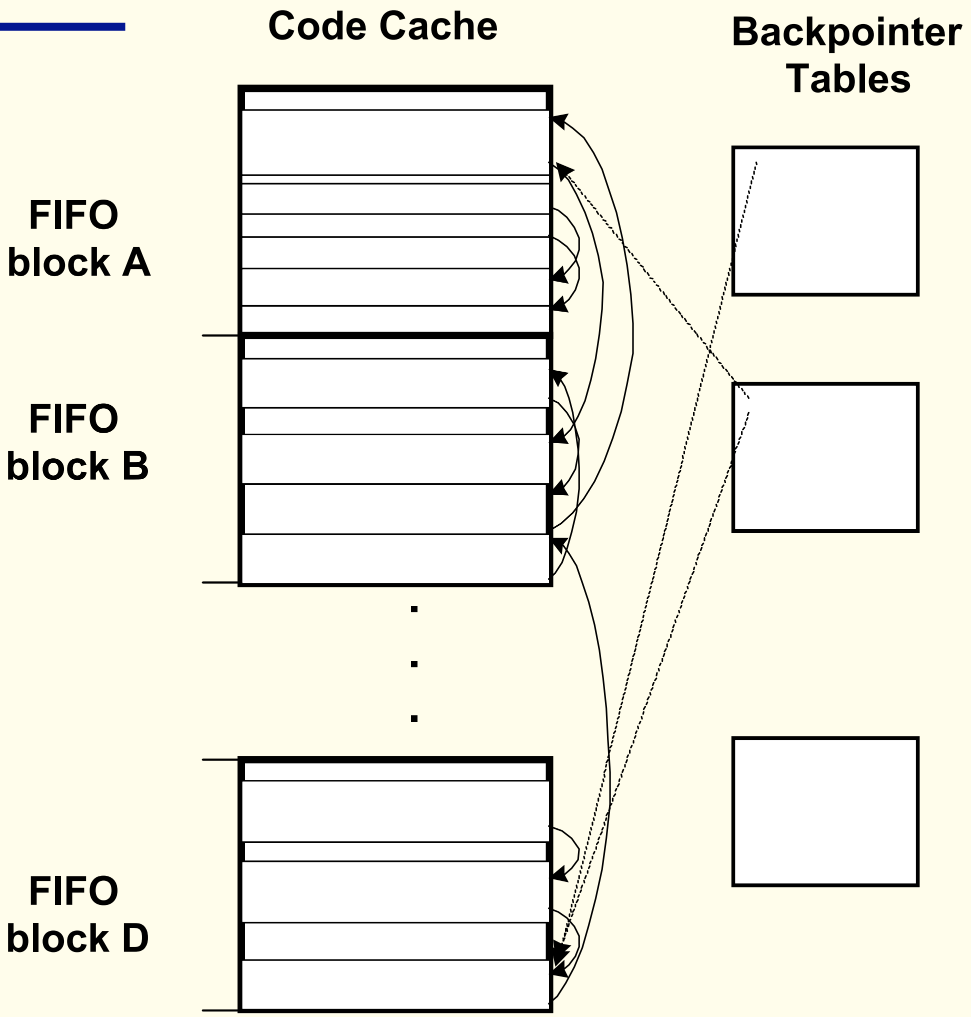
Pre-emptive Flush

- ❑ **Flush when program phase change is detected**
  - Many new translations will be needed, anyway
- ❑ **Detect when there is a burst of new translations**
- ❑ **Dynamo does this**



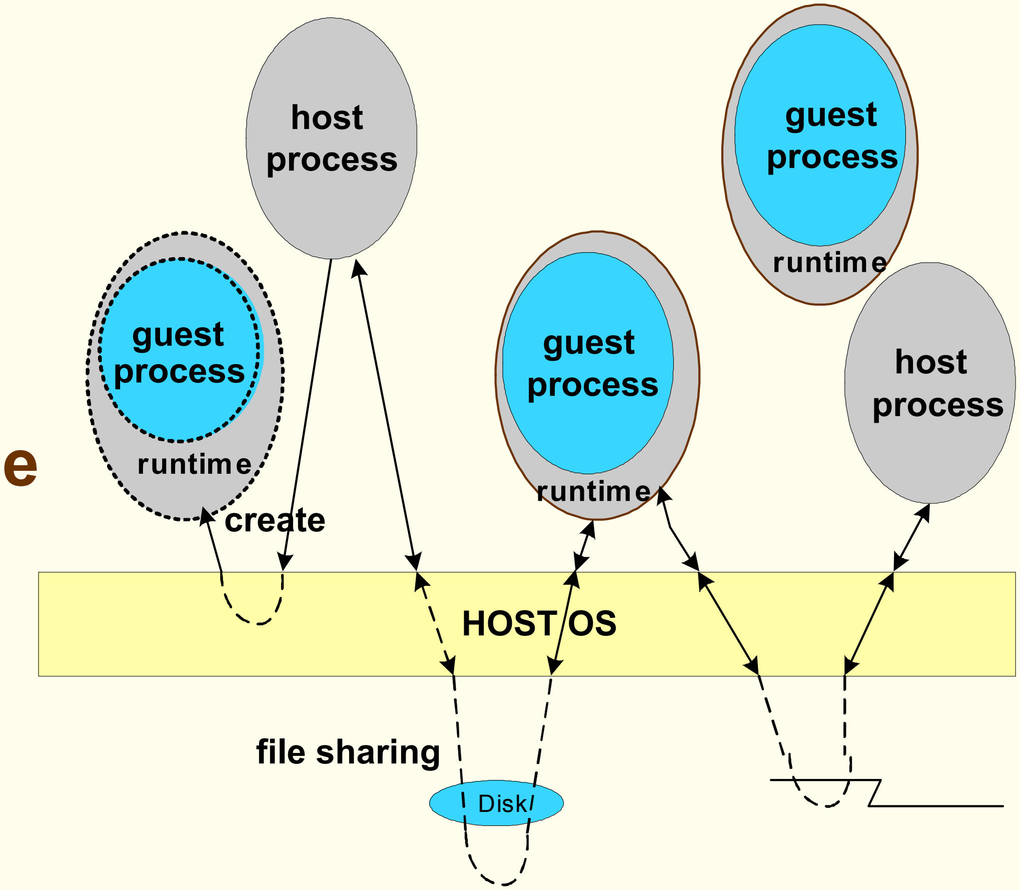
Coarse-Grain FIFO

- ❑ **Replace many blocks at once**
  - Large fixed-size blocks
  - Only backpointers among replacement blocks need to be maintained
  - OR linking between large blocks can be prohibited.



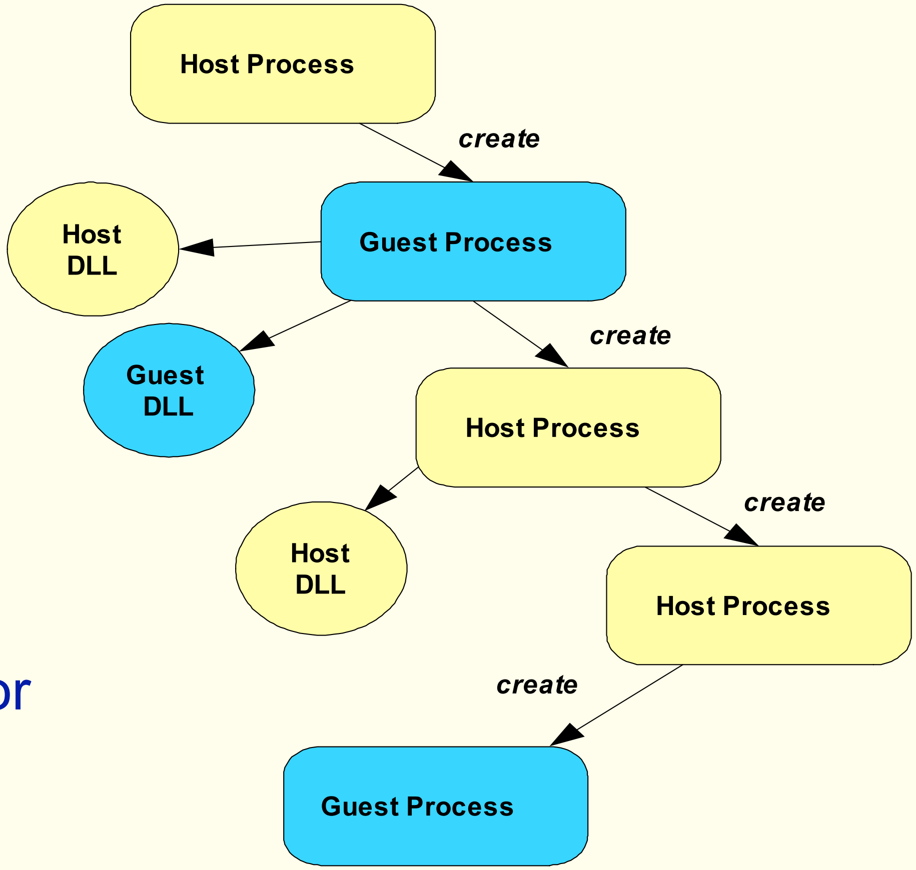
System Environment

- ❑ High level of interoperability
- ❑ Seamless access to both guest and host processes
- ❑ Works best with same OS



Encapsulation

- ❑ **Guest code is "encapsulated"**
  - At creation by loader
  - DLLs at load time
- ❑ **Creation**
  - Host can create guest
  - Guest can create host
- ❑ **DLLs**
  - Guest can use guest or host
  - Host uses only host



Loaders

- ❑ **Requires two loaders**
  - One for host processes
  - One for guest processes
- ❑ **Approaches**
  - Modify kernel loader
    - Identifies type of binary, calls correct loader
    - Requires modification of kernel loader
  - Add code to guest binary when installed
    - Invokes guest loader
    - Requires local installation of guest binary
  - Modify host process create_process API
    - Invokes guest loader for guest binaries
    - Modifies create_process in host binaries
    - Used in FX!32

Persistence

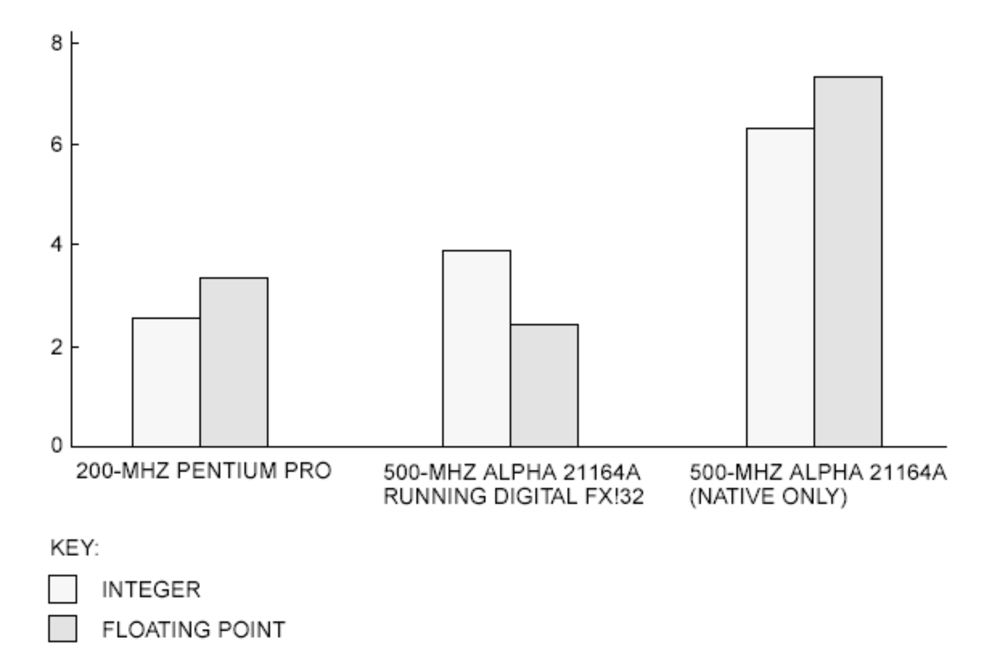
- How long do translations last?
  - One ABI instantiation
    - Re-translate each time an ABI is initiated
  - Multiple ABI instantiations
    - Save translation/profile data on disk
    - Is it faster to optimize or read from disk?
      - A lot of instructions can execute in a few milliseconds

Example: FX!32

- **x86/Windows ABIs on Alpha/Windows**
- **Runtime software**
  - Follows typical model
  - But, translations/optimizations are done between executions
    - First execution of binary: interpret and profile
    - Translate and optimize “off line”
    - Later execution(s): use translated version, continue profiling
- **Persistence**
  - Translations and profile data are saved on disk between runs
    - Very time consuming optimization with x86 source
    - Hybrid static/dynamic binary translation

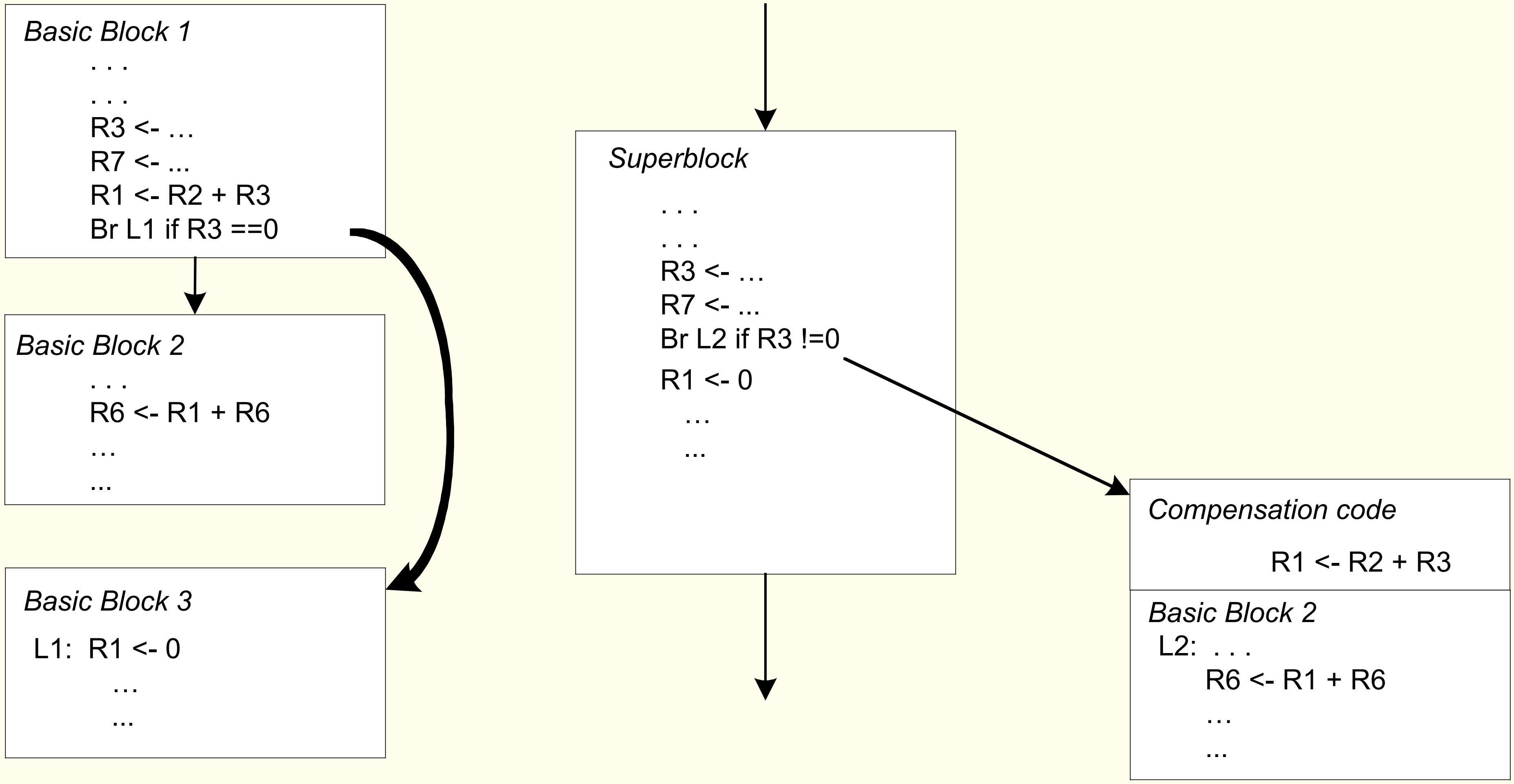
Performance

- (comparing 200 MHz Pentium Pro and 500 MHz 21164)
- Goal: same as high-end x86
- Byte benchmark integer ≈ 40% faster than Pentium Pro
- Flt point ≈ 30% slower than Pentium Pro
- Achieves 70% of native alpha performance



Dynamic Binary Optimization

Optimization Example



Basic Block 1

```
        . . .
        . . .
        R3 <- …
        R7 <- ...
        R1 <- R2 + R3
        Br L1 if R3 ==0
```

Basic Block 2

```
        . . .
        R6 <- R1 + R6
        …
        ...
```

Basic Block 3

```
 L1:  R1 <- 0
         …
         ...
```

(a)

Superblock

```
        . . .
        . . .
        R3 <- …
        R7 <- ...
        Br L2 if R3 !=0
        R1 <- 0
          …
          ...
```

Compensation code

```
             R1 <- R2 + R3
```

Basic Block 2

```
 L2:  . . .
      R6 <- R1 + R6
      …
      ...
```

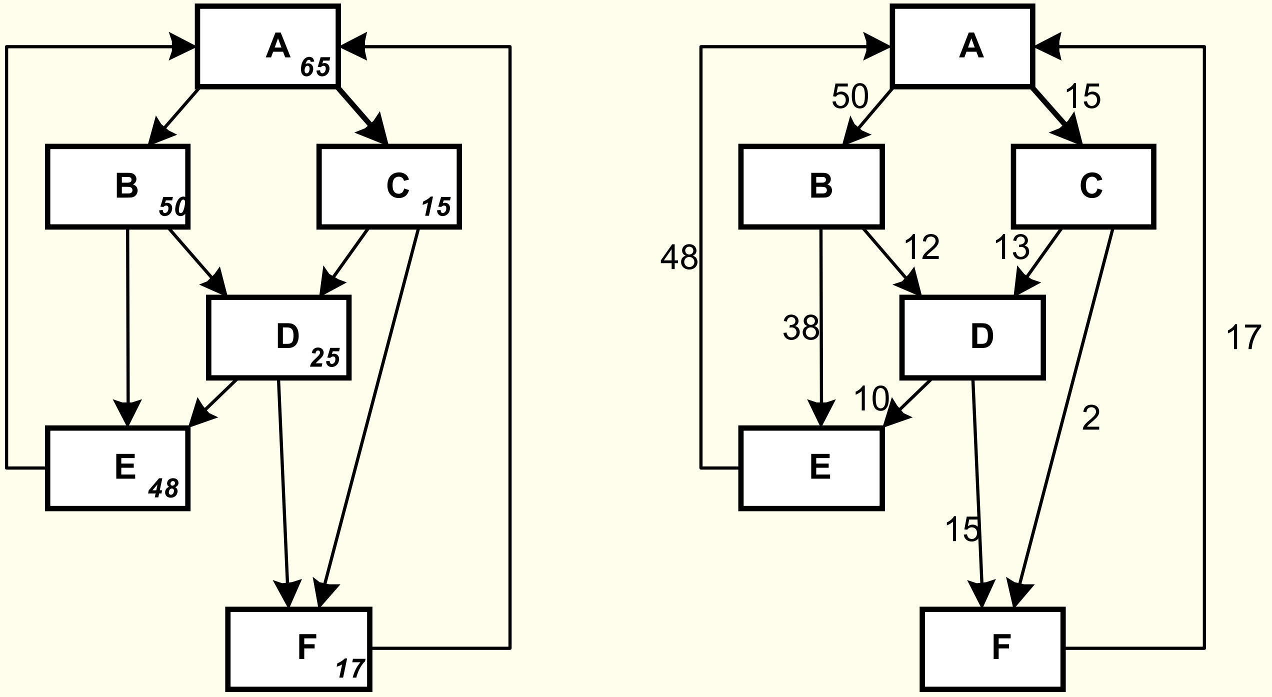
(b)

Profiling

- ❑ **Collect statistics about a program as it runs**
  - Branches (taken, not taken)
  - Jump targets
  - Data values
- ❑ **Predictability allows these statistics to be used for optimizations to be used in the future**
- ❑ **Profiling in a VM differs from traditional profiling used for compiler feedback**
  - E.g. can’t do overall analysis before inserting probes

Types of Profiles

- ❑ **Block or node profiles**
  - Identify “hot” code blocks
  - Fewer nodes than edges
- ❑ **Edge profiles**
  - Give a more precise idea of program flow
  - Block profile can be derived from edge profile (not vice versa)

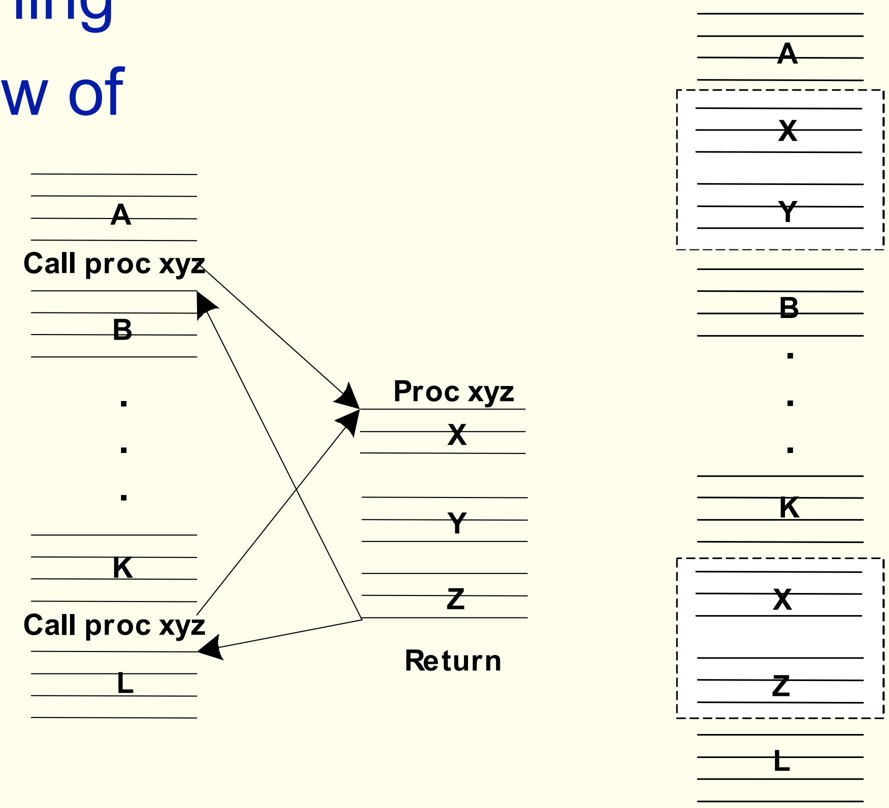


Collecting Profiles

- **Instrumentation-based**
  - Software probes
    - Slows down program more
    - Requires less total time
  - Hardware probes
    - Less overhead than software
    - Less well-supported in processors
    - Typically event counters
- **Sampling based**
  - Interrupt at random intervals and take sample
    - Slows down program less
    - Requires longer time to get same amount of data
  - Not useful during interpretation

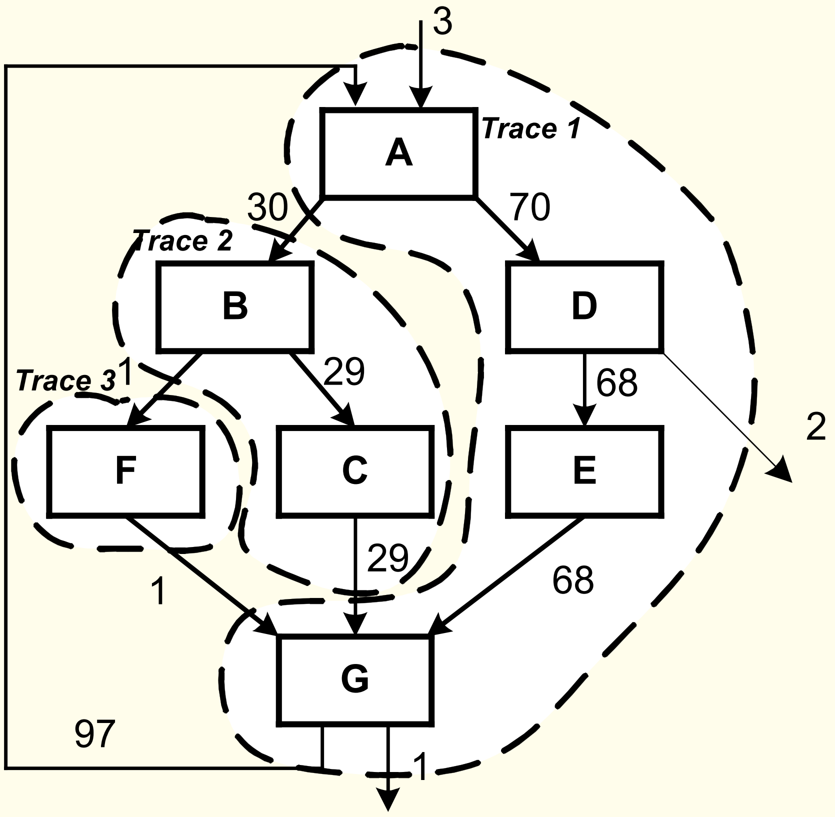
Improving Locality: Procedure Inlining

- User partial inlining
  - Unlike static full inlining
  - Follow dominant flow of control



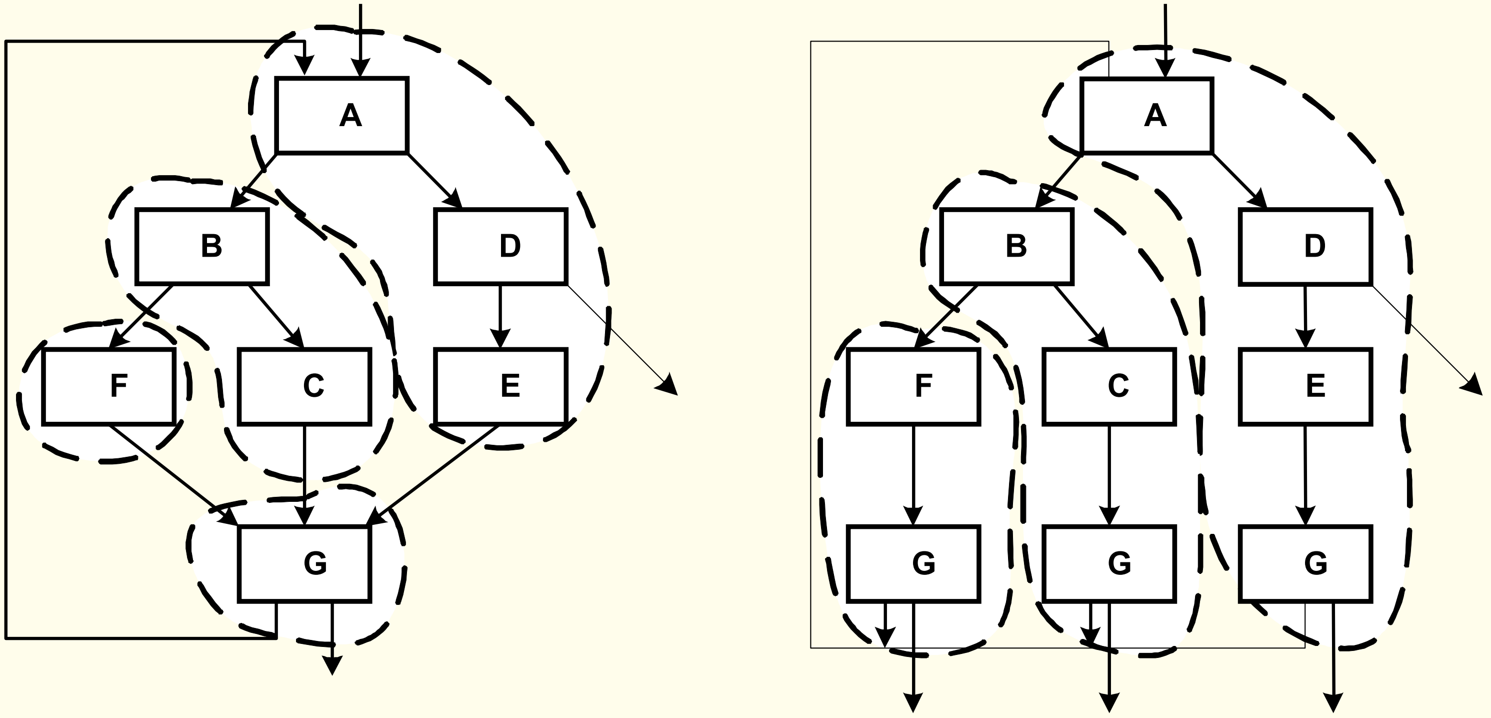
Improving Locality: Traces

- ❑ **Proposed by Fisher (Multiflow)**
  - Used overall profile/analysis
- ❑ **Join points sometimes inhibit optimizations**
- ❑ **Join points detected incrementally**
  - ⇒ bookkeeping
- ❑ **Typically not used in optimizing VMs**



Improving Locality: Superblocks

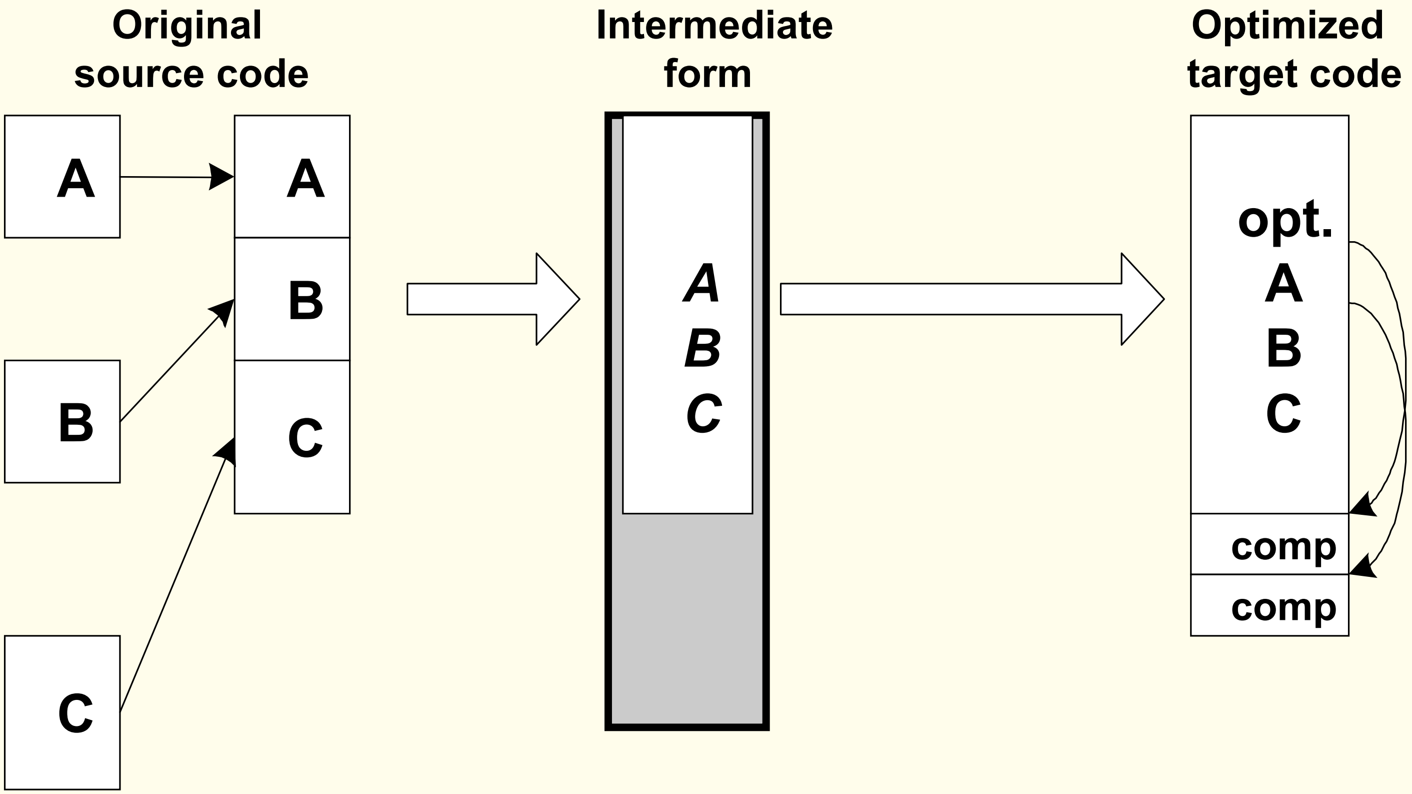
- ❑ One entry multiple exits
- ❑ May contain redundant blocks (tail duplication)
- ❑ Commonly used in optimizing VMs



Superblock Formation

- **Start Points**
  - When block use reaches a threshold
  - Profile all blocks (UQDBT)
  - Profile selected blocks (Dynamo)
    - Profile only targets of backward branches (close loops)
    - Profile exits from existing superblocks
- **Continuation**
  - Use hottest edges above a threshold (UQDBT)
  - Follow current control path (most recent edge) (Dynamo)
- **End Points**
  - Start point of this superblock
  - Start point of some other superblock
  - When a maximum size is reached
  - When no edge above threshold can be found (UQDBT)
  - When an indirect jump is reached (depends on whether inlining is enabled)

Dynamic Optimization Overview



Original source code

A

B

C

A

B

C

Intermediate form

A
B
C

Optimized target code

opt.
A
B
C

comp

comp

Collect basic blocks using profile information

Convert to intermediate form; place in buffer

Schedule and optimize

Generate target code

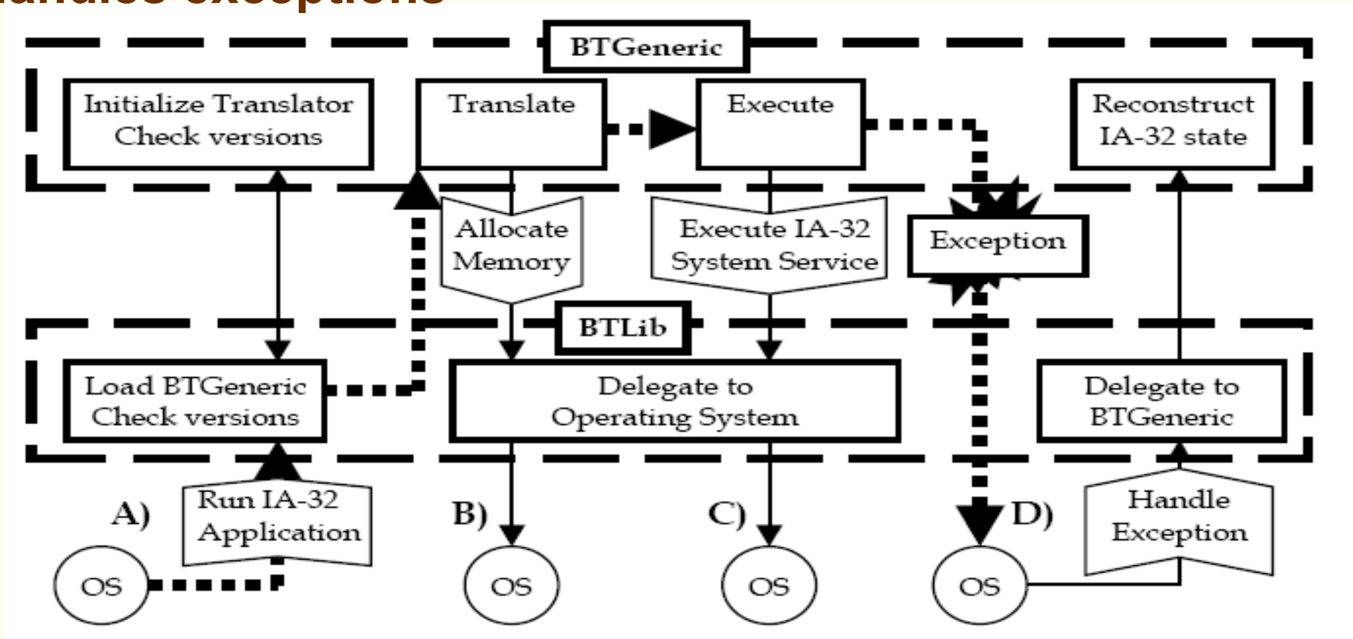
Add compensation code; place in code cache

Case Study: Intel IA-32 EL

- Software method for running IA-32 binaries on IPF
  - Previous approach was in hardware
- Runs with both Windows and Linux
  - OS independent section (BTgeneric)
  - OS dependent section (BTlib)
- Two stages
  - Fast binary translation (cold code)
  - Optimized binary translation (hot code)
- Precise traps are an important consideration

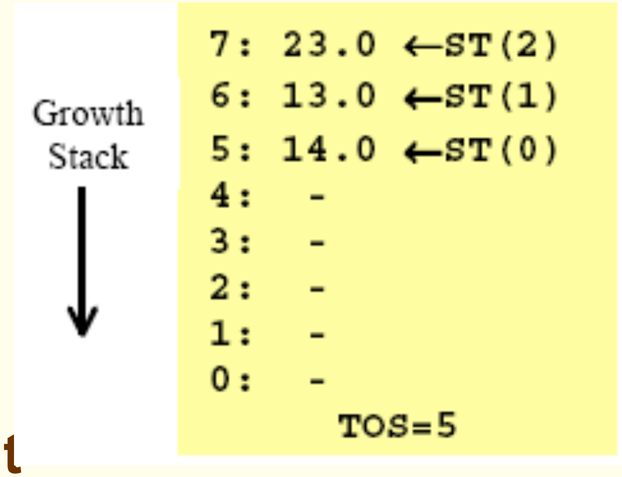
Operating System Interaction

- Process level runtime
- Supports Windows and Linux
  - BTlib is implementation dependent part
  - About 1% of total code
- Initializes structures
- Translates OS calls
- Handles exceptions



IA-32 Optimizations

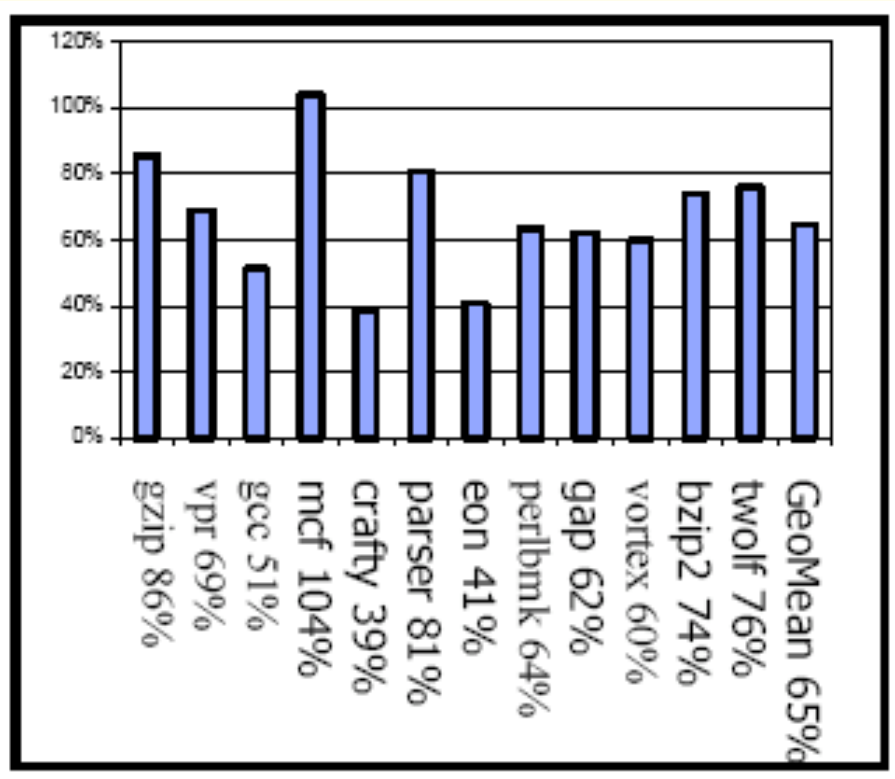
- Floating point/MMX
  - IPF uses large flat register file
  - IA-32 uses stack register file
  - IA-32 TAG indicates valid entries
  - IA-32 aliases MMX regs to FP regs



- Speculate common case usage and put guard code at beginning of block
- Examples:
  - TOS (Top of Stack) same for all block executions
  - No invalid accesses (indicated by TAG)
  - 99-100% accurate
- Data Misalignment
  - Similar to FX!32
  - See paper for details

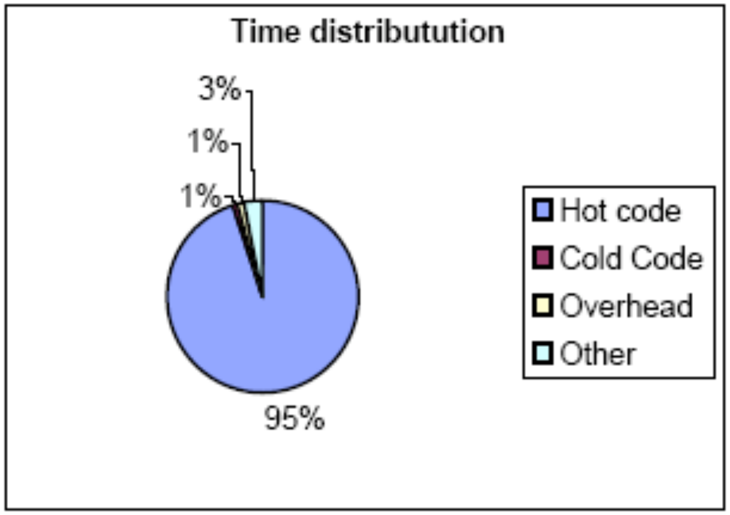
IA-32 EL Performance

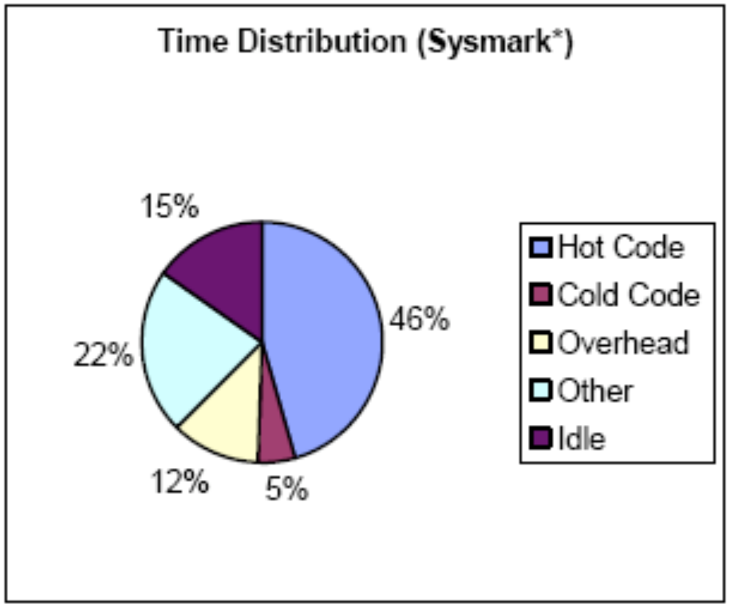
- **Comparison with native IPF performance**
  - Provides 65% performance (Gmean)
  - mcf performs better because it has a 32-bit data footprint rather than 64-bits



IA-32 EL Performance

- **Where the time is spent**
  - SPEC – mostly in hot code; very little overhead
  - Sysmark – only 45% hot code; 22% in OS (IPF code)



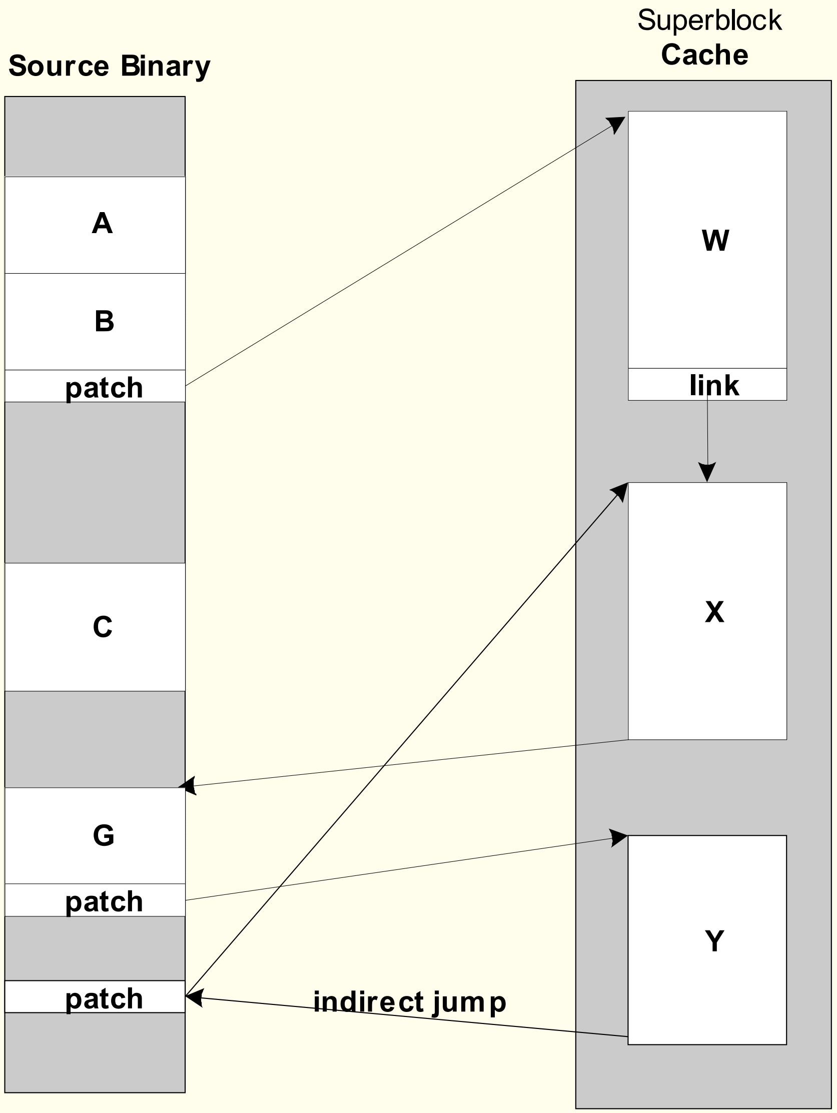


Same-ISA Optimization

- ❑ **Objective: optimize binaries on-the-fly**
  - Many binaries are un-optimized or are at a low optimization level
- ❑ **Initial emulation can be done very efficiently**
  - "Translation" at basic block level is identity translation
  - Initial sample-based profiling is attractive

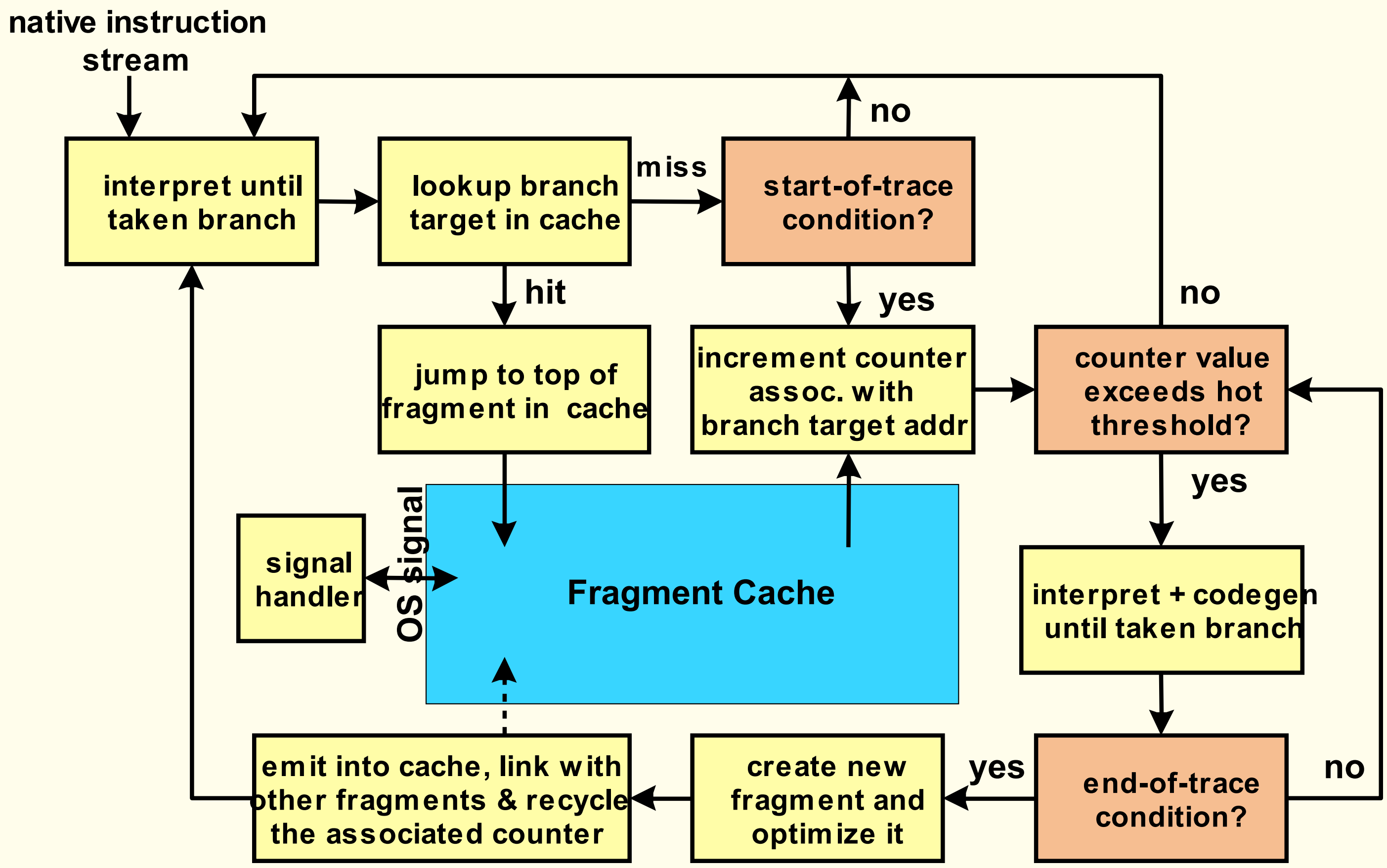
    Original code can be used, running at native speeds
- ❑ **Code patching can be used**
  - Patch code cache regions into original code
  - Replace original code with branches into code cache (saves code some code duplication)
  - Can avoid hash table lookup on indirect jumps
- ❑ **Can bail-out if performance is lost**

Code Patching Example



Case Study: HP Dynamo

- Maps HP-PA ISA onto itself
- Improved optimization is goal


native instruction stream
interpret until taken branch
lookup branch target in cache
miss
start-of-trace condition?
no
hit
yes
jump to top of fragment in cache
increment counter assoc. with branch target addr
counter value exceeds hot threshold?
no
yes
signal handler
OS signal
Fragment Cache
interpret + codegen until taken branch
emit into cache, link with other fragments & recycle the associated counter
create new fragment and optimize it
yes
end-of-trace condition?
no

Superblock Selection

- **Does not use hardware counters, PC sampling, or path sampling**
- **Interpreter performs MRET**
  - Most Recently Executed Tail
  - Associate a counter with superblock-start points
  - If counter exceeds threshold then trigger instruction collection
  - At superblock-end, collected instructions are “hot superblock”
  - Concept: when an instruction becomes hot, the very next sequence will also be hot
  - Simple, small counter overhead
- **No profiling on fragments**
  - No overheads
  - Problem if branch behavior changes
  - Fragment cache is occasionally flushed…

Prototype Implementation

- **Conservative optimizations**
  - Allow recovery of state for synchronous traps
- **Aggressive optimizations**
  - Do not allow recovery of state
  - Include –

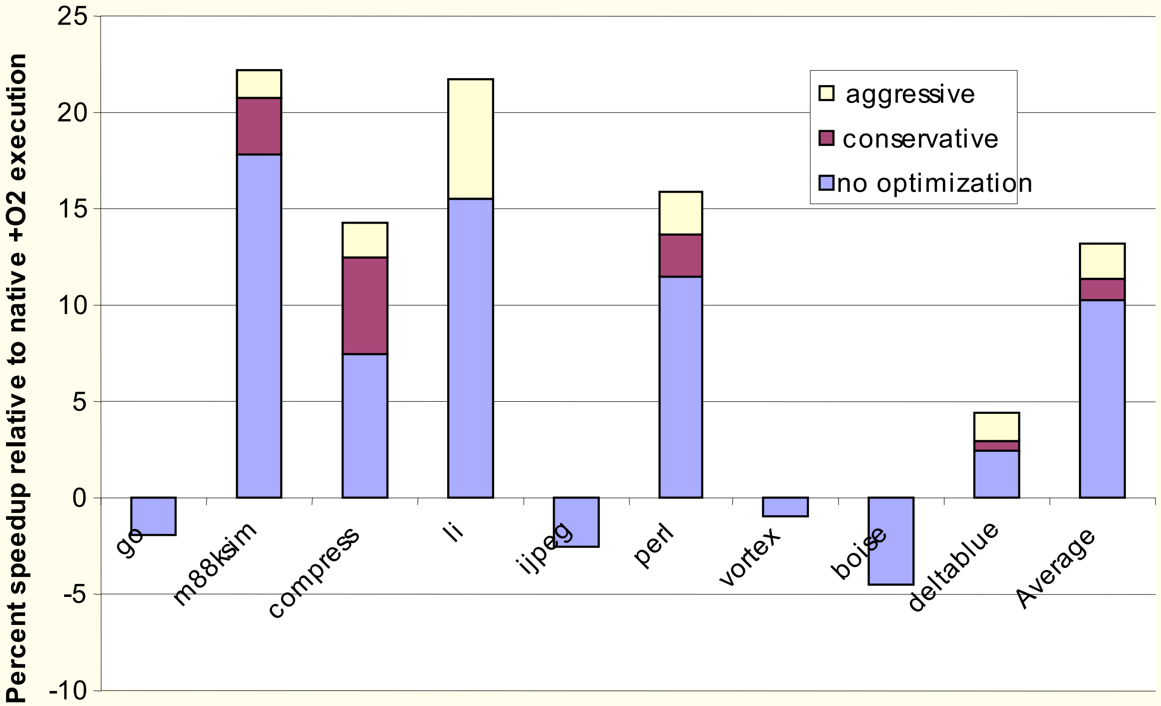
    Dead code removal

    Code sinking

    Loop invariant code motion
- **Start in aggressive mode, switch to conservative mode if "suspicious" code sequence is encountered**
- **Bail out for ill-behaved code**
  - Unstable working sets
  - Thrashing in the Fragment Cache

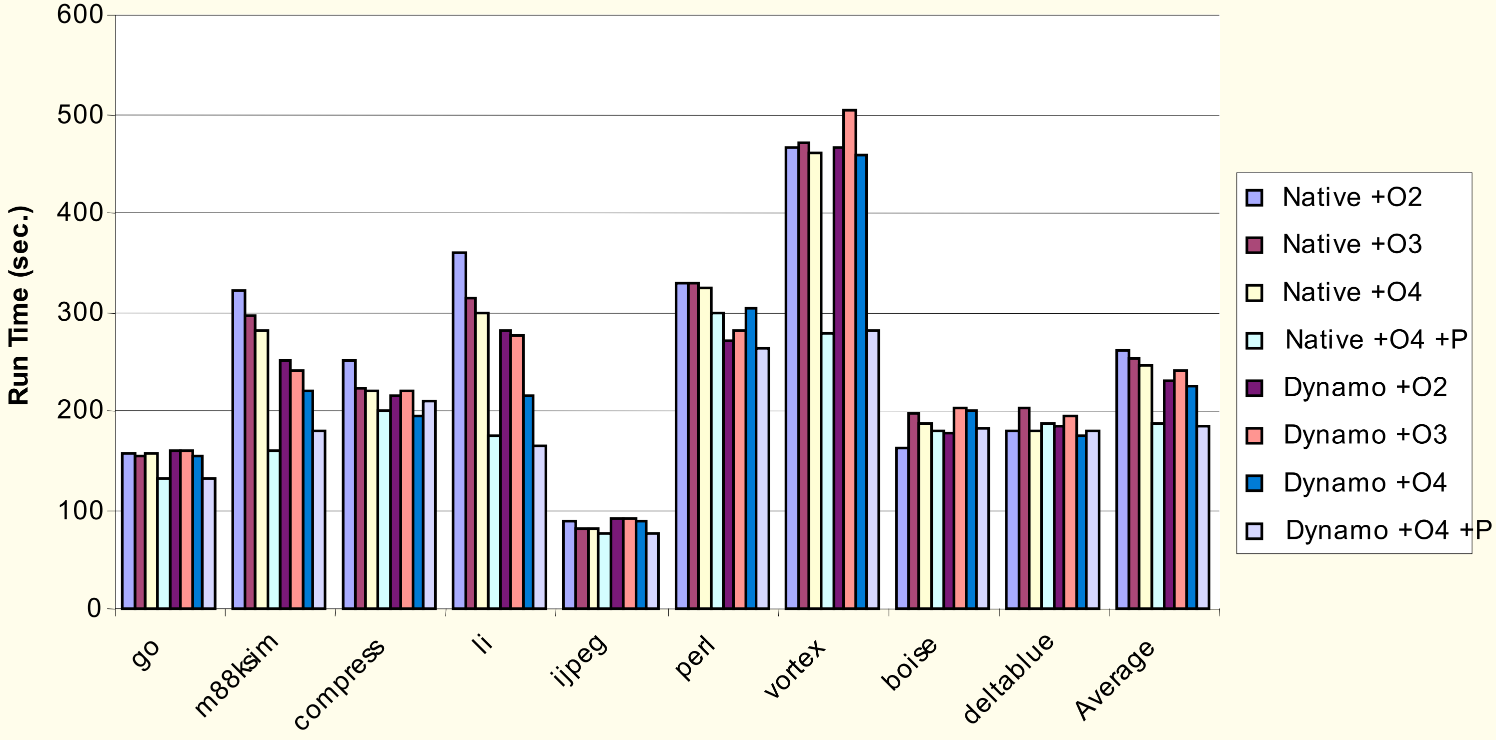
Performance

- Compare with +o2
- Biggest gain from inlining and improved code layout
- Conservative opts help about as much as aggressive
- Some benchmarks "bail-out"



Performance

- Outperforms +O2; +O4, but not +O4 plus profiling
  - This may be due to code layout
  - Many app developers do not profile



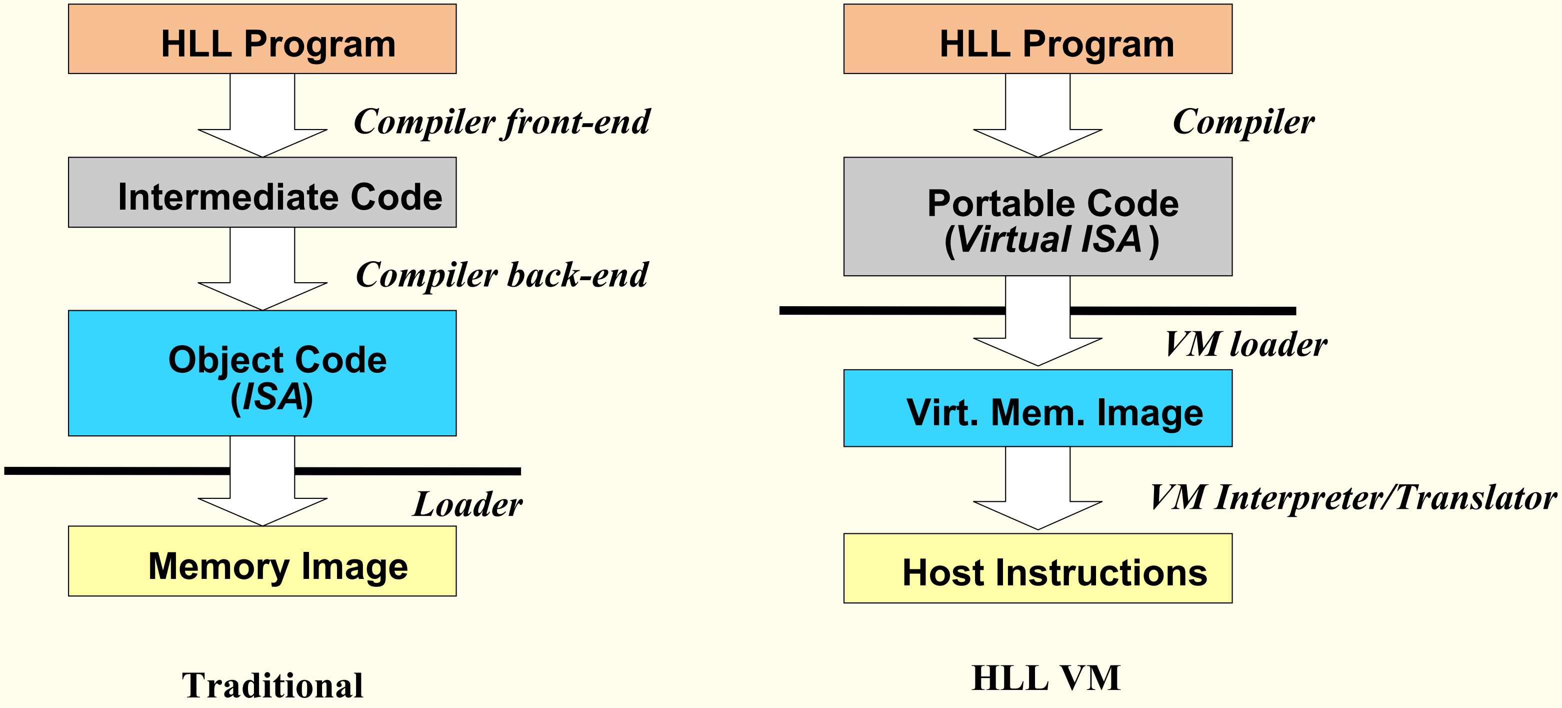
Performance Conclusions

- **Mostly useful for code optimized at low levels**
- **Dynamo ran on processor that stalled indirect jumps**
  - Baseline is slow compared with most superscalar processors
  - Dynamo removes indirect jumps via procedure inlining and inlined software jump prediction
- **On other modern processors there is a significant performance loss due to indirect jumps**
  - See Dynamo/RIO (x86)
  - RIO project targets security, not performance

High Level Language VMs

HLL VMs

- ❑ Goal: complete platform independence for applications
- ❑ Similar to Process VMs
  - Major difference is specification level:
    Virtual instruction set + libraries
  - Instead of ISA and OS interface



UCSD P-Code

- ❑ Popularized HLL VMs
- ❑ Provided highly portable version of Pascal
- ❑ Consists of
  - Primitive libraries
  - Machine-independent object file format
  - Stack-based ISA
  - A set of byte-oriented “pseudo-codes”
  - Virtual machine definition of pseudo-code semantics

Modern HLL VMs

- **Superficially similar to P-code scheme**
  - Stack-based ISA
  - Standard libraries

    BUT, Objective is application portability, not compiler portability
- **Network Computing Environment**
  - Untrusted software (this is the internet, after all)
  - Robustness (generally a good idea)

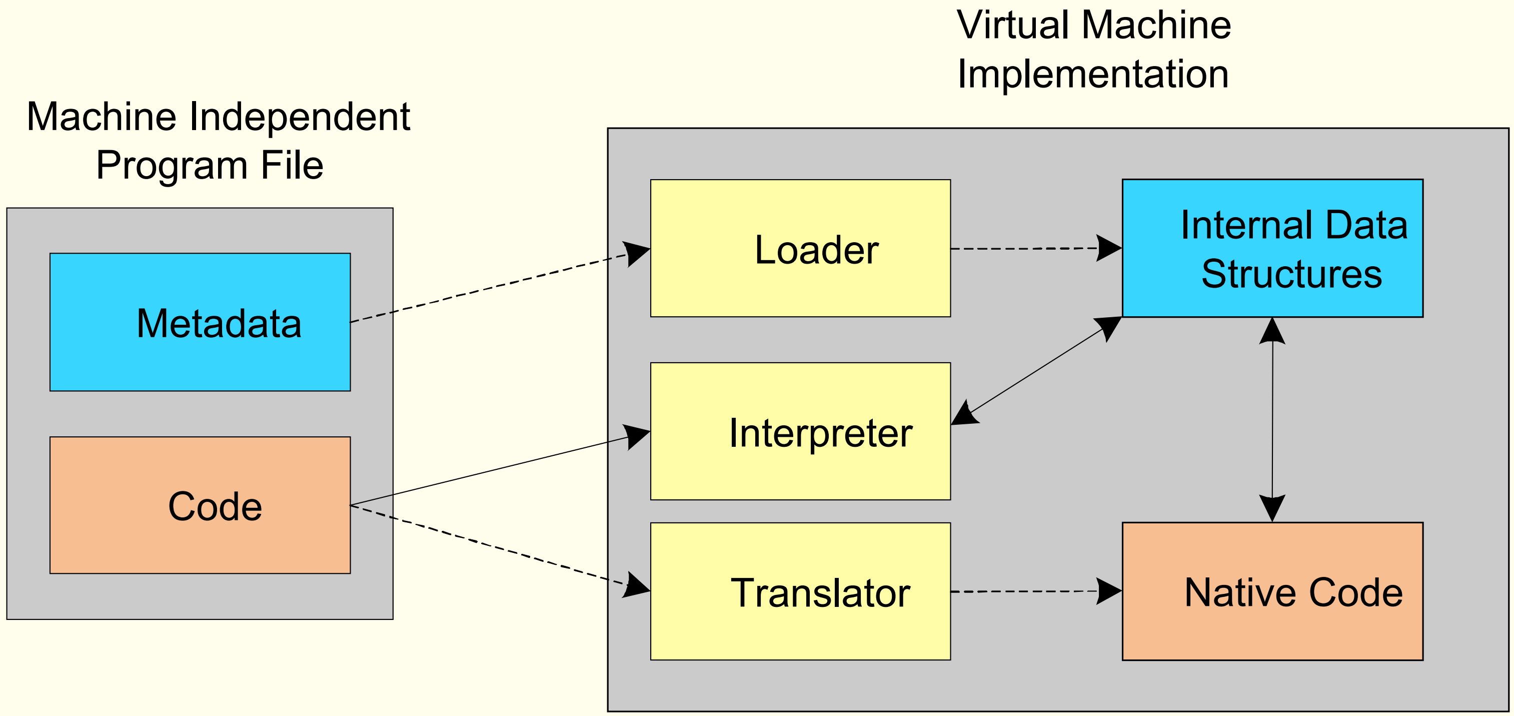
    => object-oriented programming
  - Bandwidth is a consideration
  - Good performance must be maintained
- **Two major examples**
  - Java VM
  - Microsoft Common Language Infrastructure (CLI)

Terminology

- ❑ **Java Virtual Machine Architecture ⇔ CLI**
  - Analogous to an ISA
- ❑ **Java Virtual Machine Implementation ⇔CLR**
  - Analogous to a computer implementation
- ❑ **Java bytecodes ⇔ Microsoft Intermediate Language (MSIL), CIL, IL**
  - The instruction part of the ISA
- ❑ **Java Platform ⇔ .NET framework**
  - ISA + Libraries; a higher level ABI

Modern HLL VMs

- Compiler forms program files (e.g. class files)
  - Standard format
  - In theory any compiler can be used
- Program files contain both code and metadata

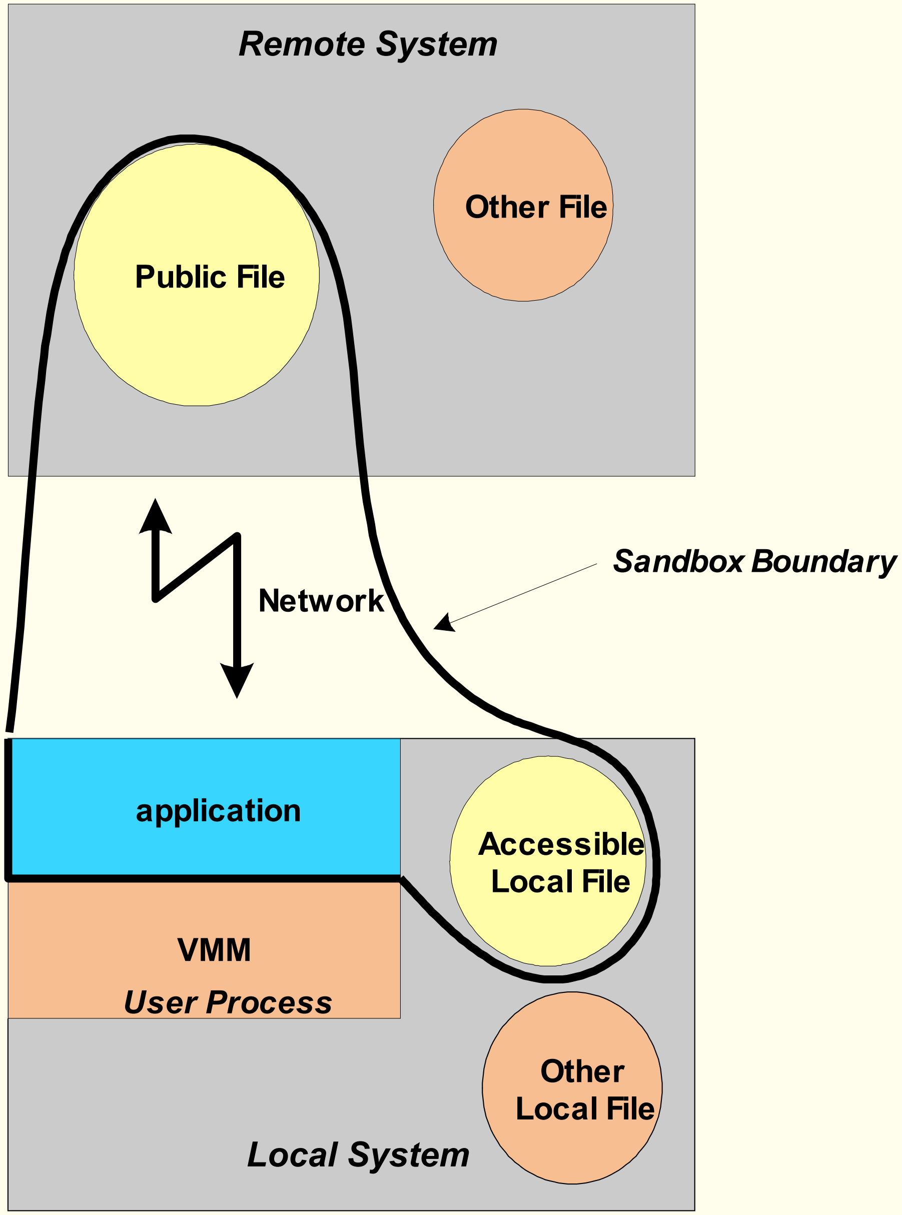


Robustness: Object-Orientation

- **Objects**
  - Data carrying entities
  - Dynamically allocated
  - Must be accessed via pointers or *references*
- **Methods**
  - Procedures that operate on objects
  - Method operating on an object is like “sending a message”
- **Classes**
  - A *type* of object and its associated methods
  - Object created at runtime is an *instance* of the class
  - Data associated with a class may be *dynamic* or *static*

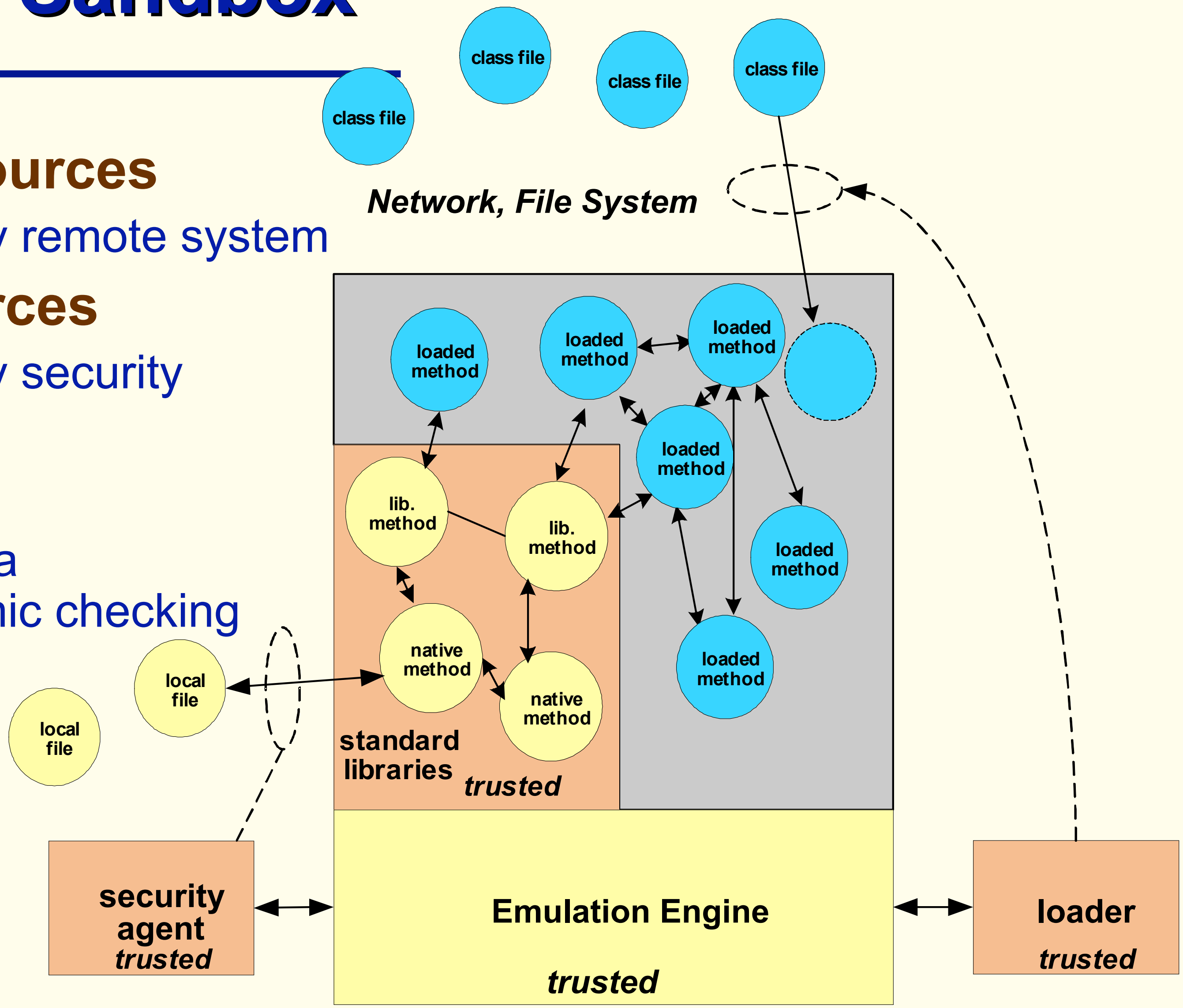
Security

- A key aspect of modern network-oriented VMs
- Rely on “protection sandbox”
- Must protect:
  - Remote resources (files)
  - Local files
  - Runtime from user process
- This is the first generation security method – still the default



Protection Sandbox

- Remote resources
  - Protected by remote system
- Local resources
  - Protected by security manager
- VM software
  - Protected via static/dynamic checking



Garbage Collected Heap

- ❑ **Objects are created and “float” in memory space**
  - Tethered by references
  - In architecture, memory is unbounded in size
  - In reality it is limited
- ❑ **Garbage creation**
  - During program execution, many objects are created then abandoned (become garbage)
- ❑ **Collection**
  - Due to limited memory space, Garbage should be collected so memory can be re-used
  - Forcing programmer to explicitly free objects places more burden on programmer
    - Can lead to memory leaks, reducing robustness
  - To improve robustness, have VM collect garbage automatically

Network Friendliness

- ❑ **Support dynamic class file loading on demand**
  - Load only classes that are needed
  - Spread loading out over time
- ❑ **Compact instruction encoding**
  - Use stack-oriented ISA (as in Pascal)
  - Metadata also consumes bandwidth, however

    Overall, it is probably a wash

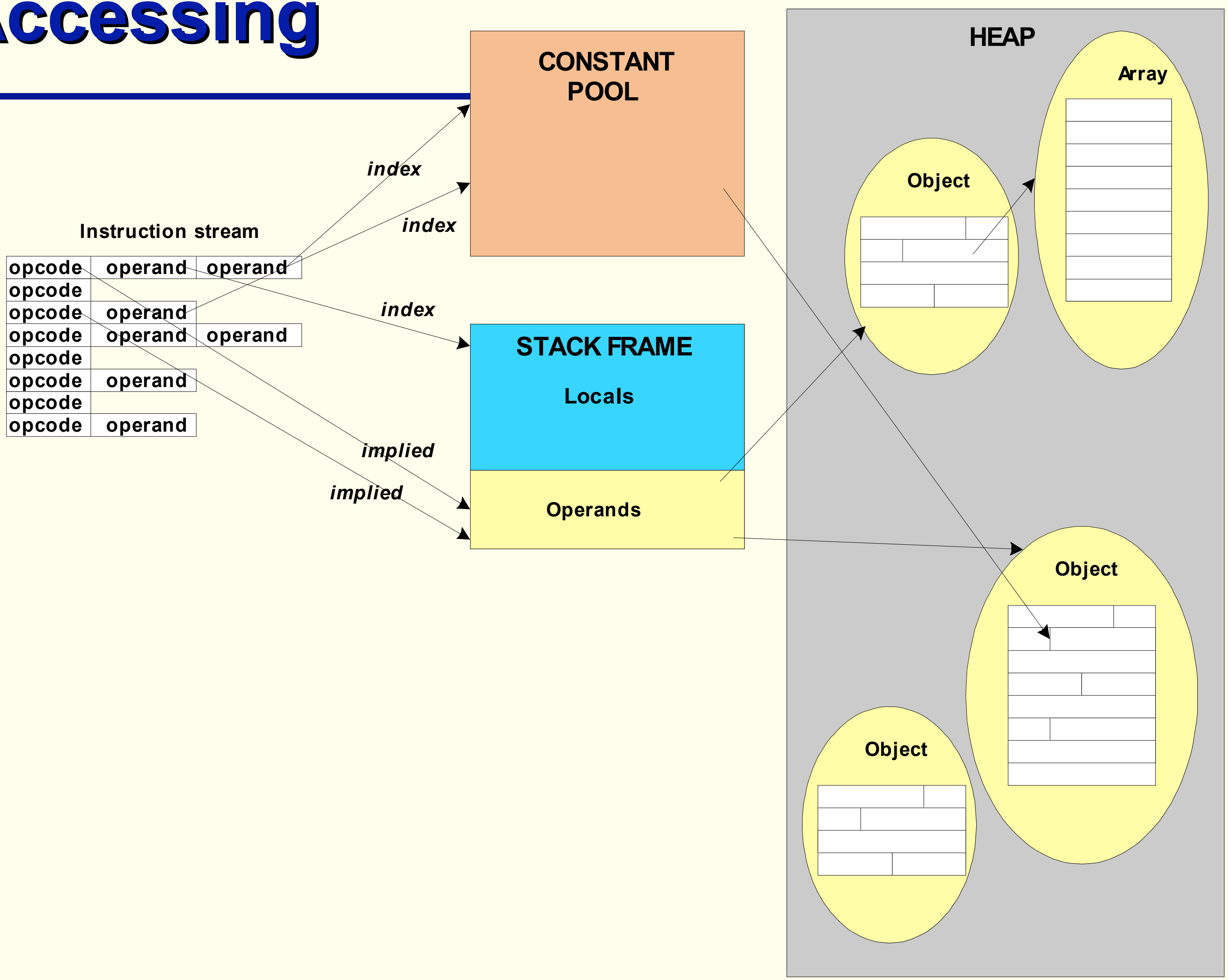
Java “ISA”

- ❑ **Includes**
  - Bytecode (instruction) definitions
  - Metadata: data definitions and inter-relationships
- ❑ **Formalized in class file specification**

Java Architected State

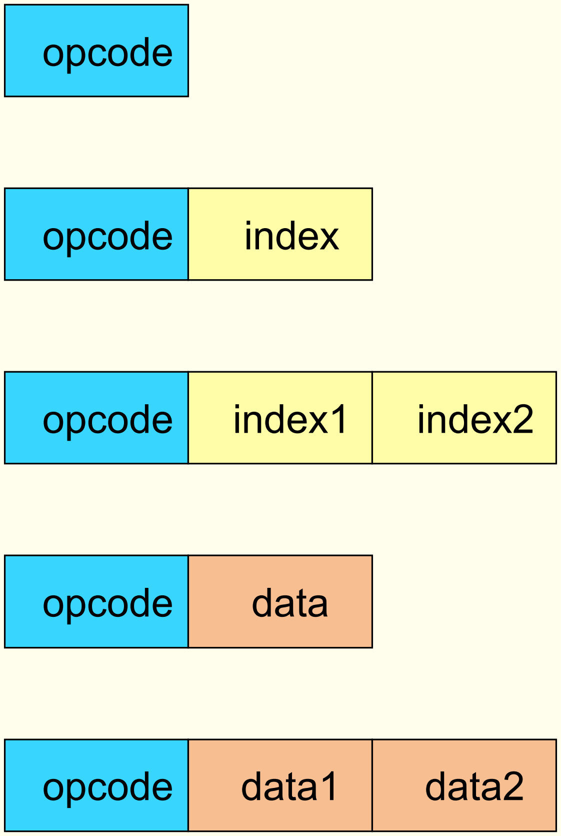
- **Implied Registers**
  - PC
  - Stack Pointer
  - etc.
- **Stack**
  - Locals
  - Operands
- **Heap**
  - Objects
  - Arrays (intrinsic objects)
- **Class file contents**
  - Constant pool holds immediates and other constant information

Data Accessing



Instruction Set

- Defined for class file, not memory image
- Bytecodes
  - One byte opcode
  - Zero or more operands
    - Opcode indicates how many
- Can take operands from
  - Instruction
  - Current constant pool
  - Current frame local variables
  - Values on operand stack
    - Distinguish storage types and computation types



opcode

opcode | index

opcode | index1 | index2

opcode | data

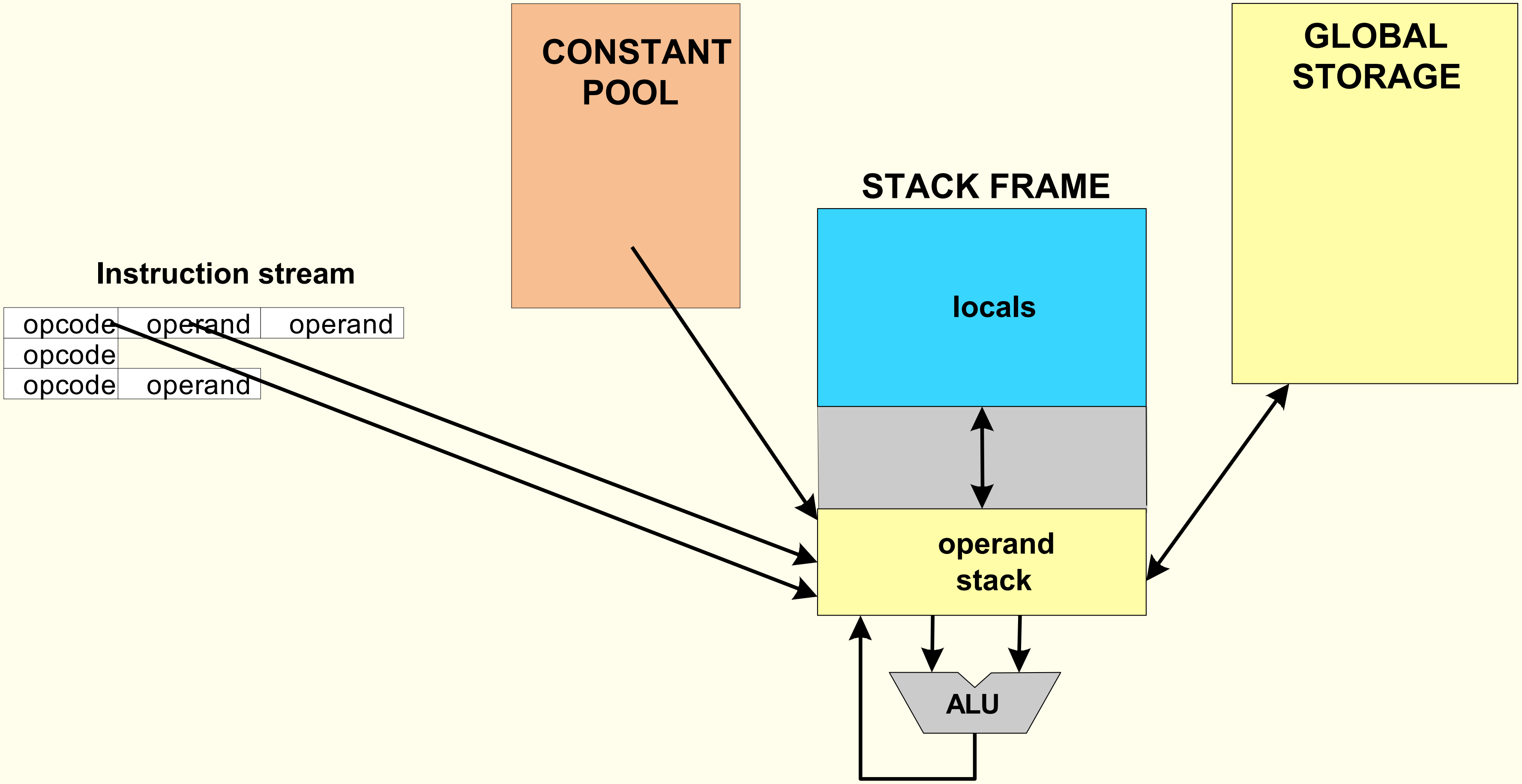
opcode | data1 | data2

Instruction Types

- Pushing constants onto the stack
- Moving local variable contents to and from the stack
- Managing arrays
- Generic stack instructions (dup, swap, pop & nop)
- Arithmetic and logical instructions
- Conversion instructions
- Control transfer and function return
- Manipulating object fields
- Method invocation
- Miscellaneous operations
- Monitors

Data Movement

- All data movement takes place through stack



CONSTANT POOL

GLOBAL STORAGE

STACK FRAME

locals

Instruction stream

opcode	operand	operand
opcode		
opcode	operand	

operand stack

ALU

Bytecode Example

```
public int perimeter();
 PC      instruction
    0:     iconst_2
    1:     aload_0
    2:     getfield   #2;    //Field: sides reference
    5:     iconst_0
    6:     iaload
    7:     aload_0
    8:     getfield   #2;    //Field: sides reference
   11:     iconst_1
   12:     iaload
   13:     iadd
   14:     imul
   15:     ireturn
```

Stack Tracking

- ❑ **Operand stack at any point in program has:**
  - Same number of operands
  - Of same types
  - In same order

  Regardless of control flow path getting there
- ❑ **Helps with static type checking**

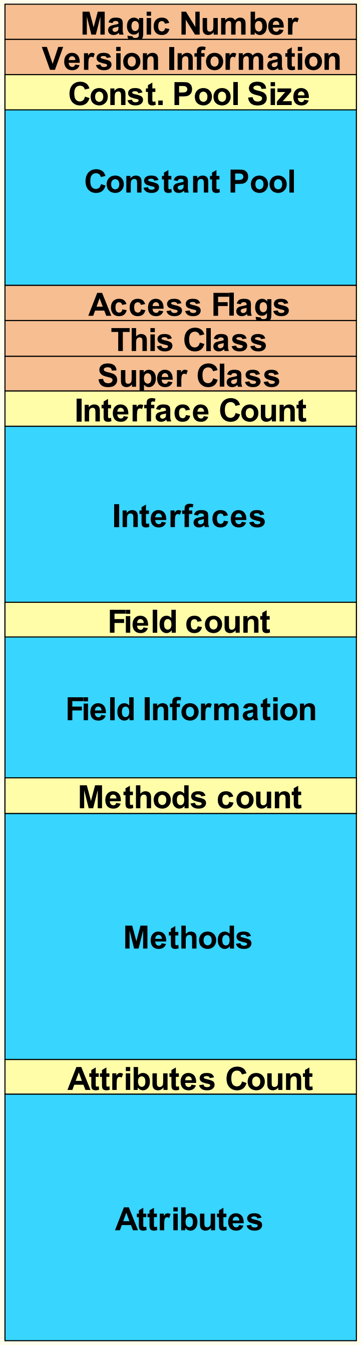
Exception Table

- Exceptions identified by table in class file
- Address Range where checking is in effect
- Target if exception is thrown
  - Operand stack is emptied
- If no table entry in current method
  - Pop stack frame and check calling method
  - Default handlers at main

From	To	Target	Type
8	12	96	**Arithmetic Exception**

Binary Classes

- ❑ **Formal ISA Specification**
- ❑ **Magic number and header**
- ❑ **Major regions preceded by counts**
  - Constant pool
  - Interfaces
  - Field information
  - Methods
  - Attributes

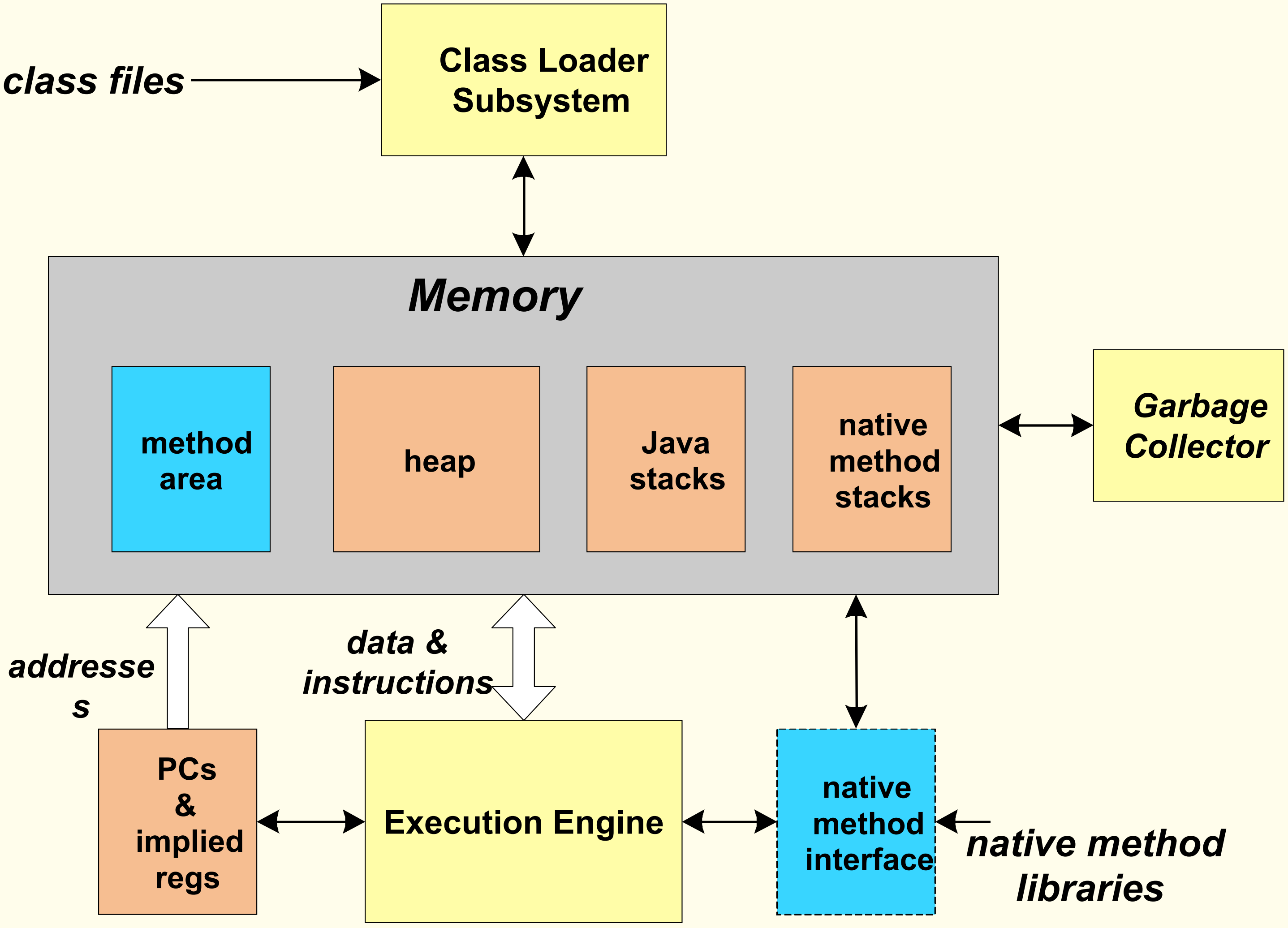


Magic Number
Version Information
Const. Pool Size
Constant Pool
Access Flags
This Class
Super Class
Interface Count
Interfaces
Field count
Field Information
Methods count
Methods
Attributes Count
Attributes

Java Virtual Machine

- An abstract entity that gives meaning to class files
- Has many concrete implementations
  - Hardware
  - Interpreter
  - JIT compiler
- Persistence
  - An instance is created when an application starts
  - Terminates when the application finishes

Structure of Virtual Machine



Structure of Virtual Machine, contd.

- ❑ **Method Area**
  - Type information provided by class loader
- ❑ **Heap Area**
  - Contains objects created by program
- ❑ **PC Register & Implied Registers**
  - Every created thread gets a set
- ❑ **Java stacks**
  - Every created thread gets one
  - Divided into Frames
  - Contains state of method invocations for the thread
  - Local variables, parameters, return value, operand stack
- ❑ **Native method stacks**
  - Special area for implementation-dependent native methods

Class Loader Subsystem

- ❑ **Primordial loader**
- ❑ **Other loaders that are part of apps**
- ❑ **Tasks:**
  - Finds and imports binary information describing type
  - Verifies correctness of type
  - Allocates and initializes memory for class variables
  - Resolves symbolic references to direct references
  - Invokes initialization code

Protection Sandbox: Security Manager

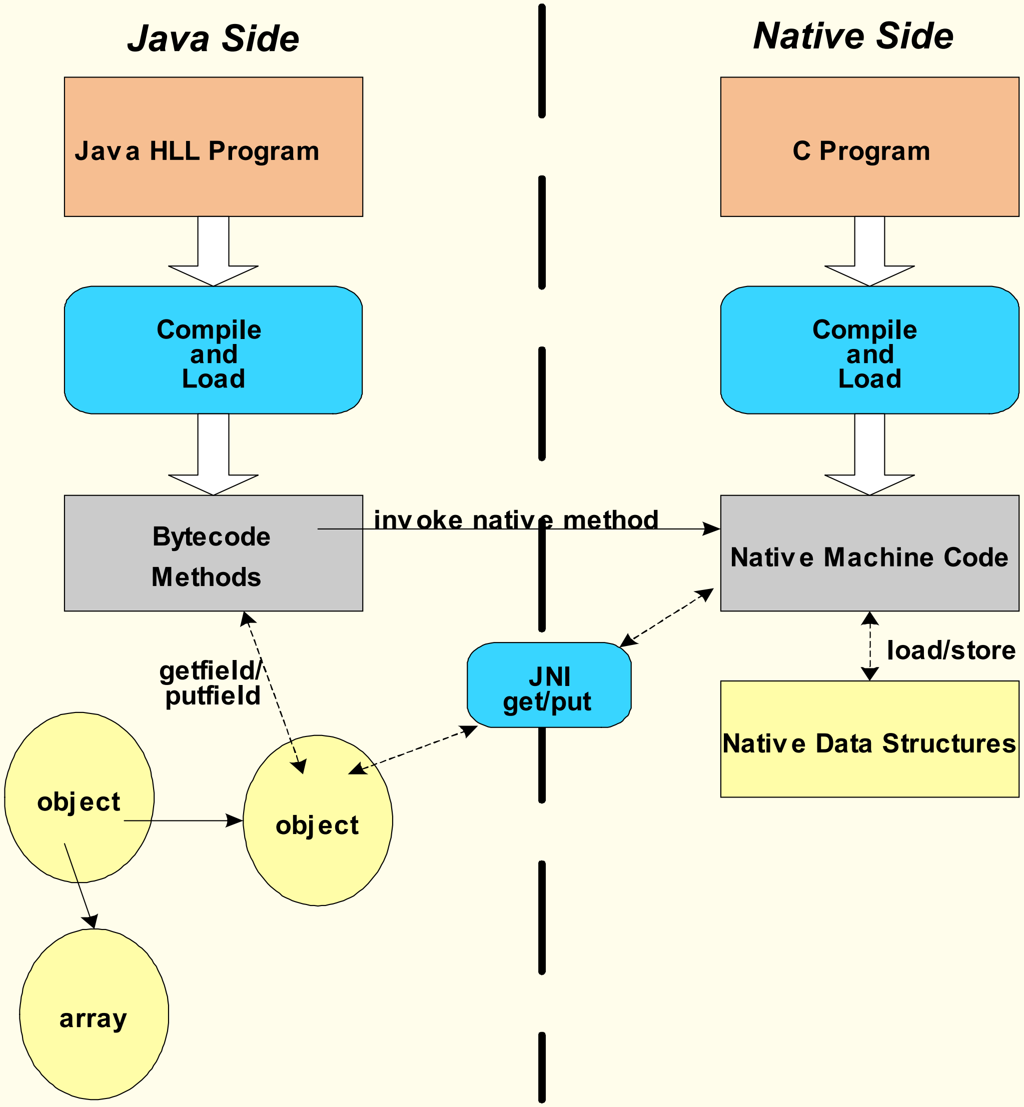
- **A trusted class containing check methods**
- **Attached when Java program starts**
  - Cannot be removed or changed
- **User specifies checks to be made**
  - Files, types of access, etc.
- **Operation**
  - Native methods that involve resource accesses (e.g. I/O) first call check method(s)

Verification

- **Class files are checked when loaded**
  - To ensure security and protection
- **Internal Checks**
  - Checks for magic number
  - Checks for truncation or extra bytes
    - Each component specifies a length
  - Make sure components are well-formed
- **Bytecode checks**
  - Check valid opcodes
  - Perform full path analysis
    - Regardless of path to an instruction contents of operand stack must have same number and types of items
    - Checks arguments of each bytecode
    - Check no local variables are accessed before assigned
    - Makes sure fields are assigned values of proper type

Java Native Interface (JNI)

- Allows Java and native SW to interoperate
  - E.g. Java can call C program (and vice versa)
  - Native routines allow access of Java data



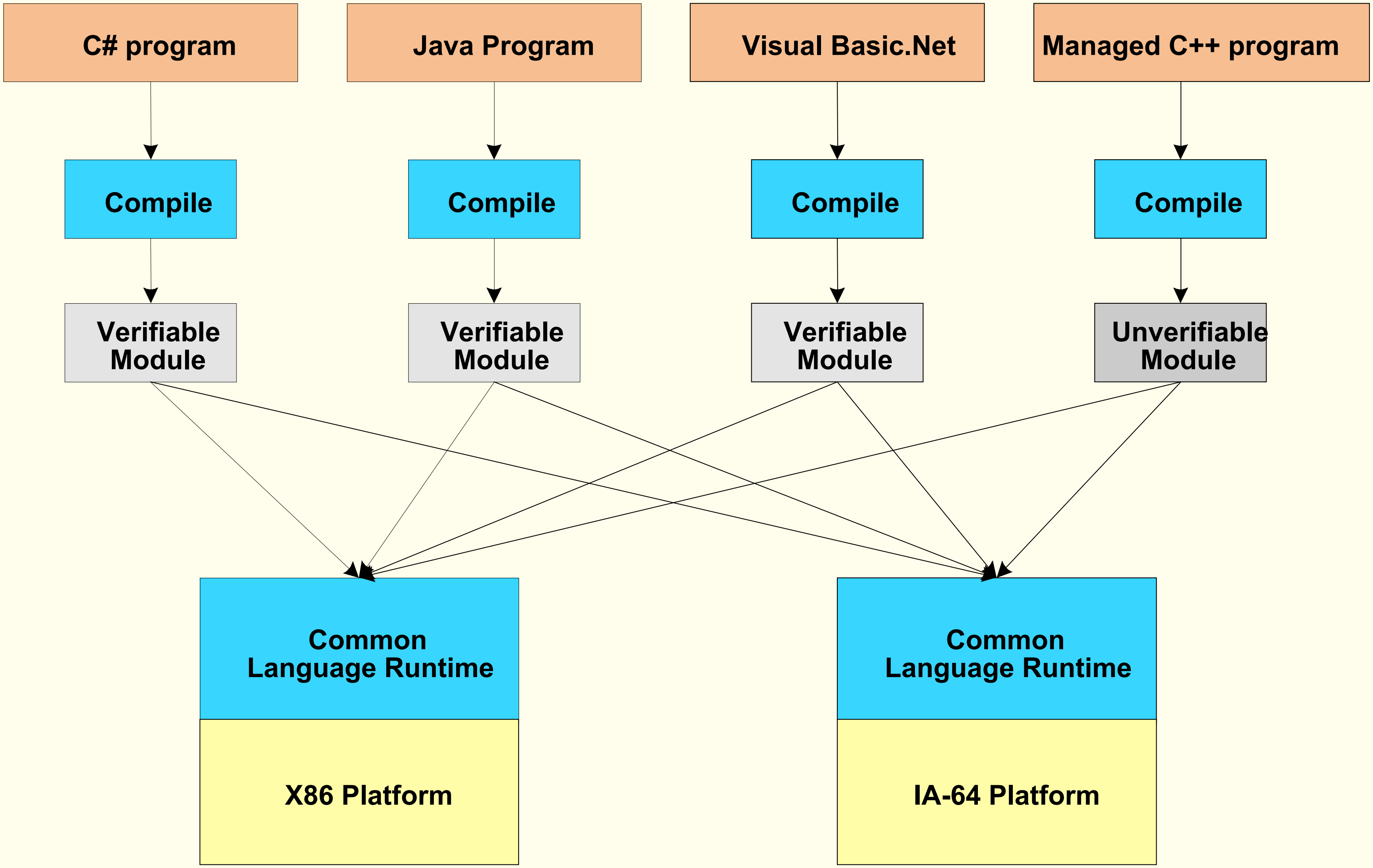
JVM Bytecode Emulation

- **Interpretation**
  - Simple, fast startup, but slow
- **Just-In-Time (JIT) Compilation**
  - Compile each method when first touched
  - Simple, static optimizations
- **Hot-Spot Compilation**
  - Find frequently executed code
  - Apply more aggressive optimizations on that code
  - Typically phased with interpretation or JIT
- **Dynamic Compilation**
  - Based on Hot-Spot compilation
  - Use runtime information to optimize

Microsoft CLI

- Common Language Infrastructure
- Part of .NET framework
- Allows multiple HLLs and multiple Platforms
- Allows both verifiable and unverifiable modules (class files)
  - Verifiability is different from validity
  - Unverifiable modules must be trusted by user
  - Verifiable and unverifiable modules can be mixed (but the program becomes unverifiable)

Microsoft CLI Interoperability



Microsoft CLI and MSIL

- Similar to Java and JVM
  - Object oriented
  - Stack-based ISA
- Some differences
  - Broader in scope
  - ISA not designed for interpretation
  - Module can be valid (but not verifiable), verifiable, or invalid

    Support for C-like pointers and un-typed memory blocks (not verifiable)

Summary: HLL VMs vs. Process VMs

- **Memory architecture**
  - Object model is less implementation-dependent
    - No compatibility problems due to size limitations/differences
- **Memory protection**
  - Pointers very carefully controlled
    - No rogue load/stores
- **Precise Exceptions**
  - Exception checking is explicit (no masks)
  - Operand stack imprecise within a method
  - Locals imprecise if exception goes to higher level

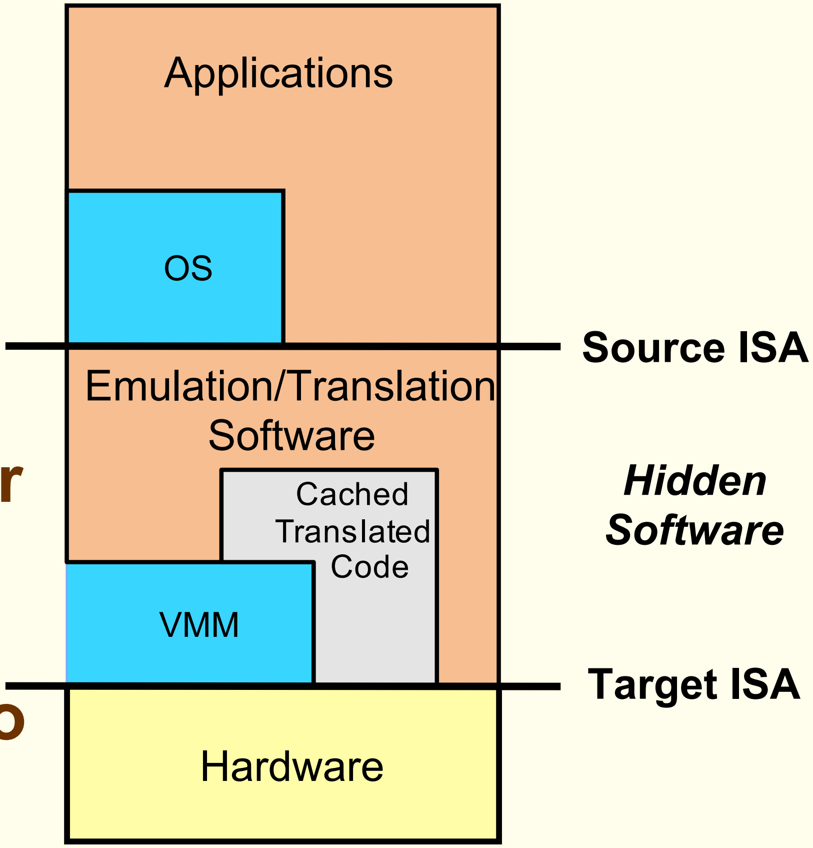
Summary: HLL VMs vs. Process VMs

- **Instruction set dependences**
  - No registers
  - No condition codes
- **Code discovery**
  - Restricted, explicit control flow
  - All code can be discovered at method entry
- **Self Modify-Referencing Code**
  - Simply doesn’t exist

Co-Designed VMs

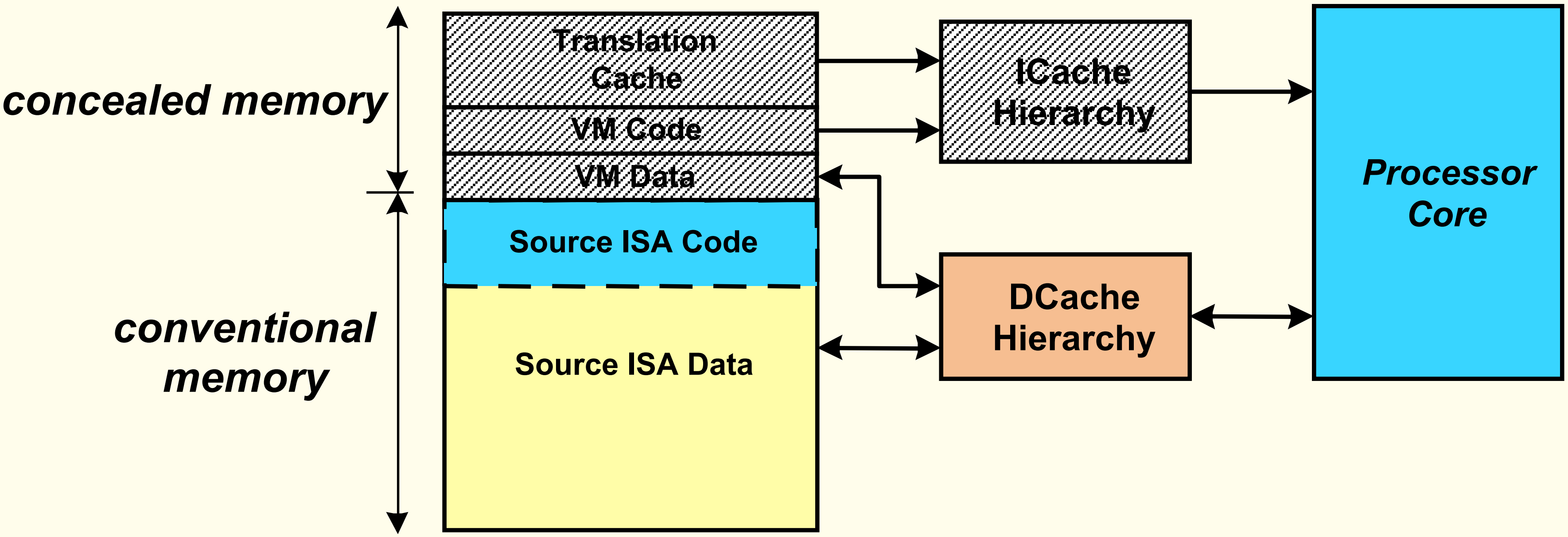
Co-Designed VMs

- Design hardware and VM software concurrently and cooperatively
- Use proprietary target ISA
  - Or modified ISA
- No native OS or applications
- Goal is performance or power efficiency
  - *Not compatibility*
- Techniques also applicable to HW support for other VMs



Concealed Memory

❑VM software resides in memory concealed from all conventional software



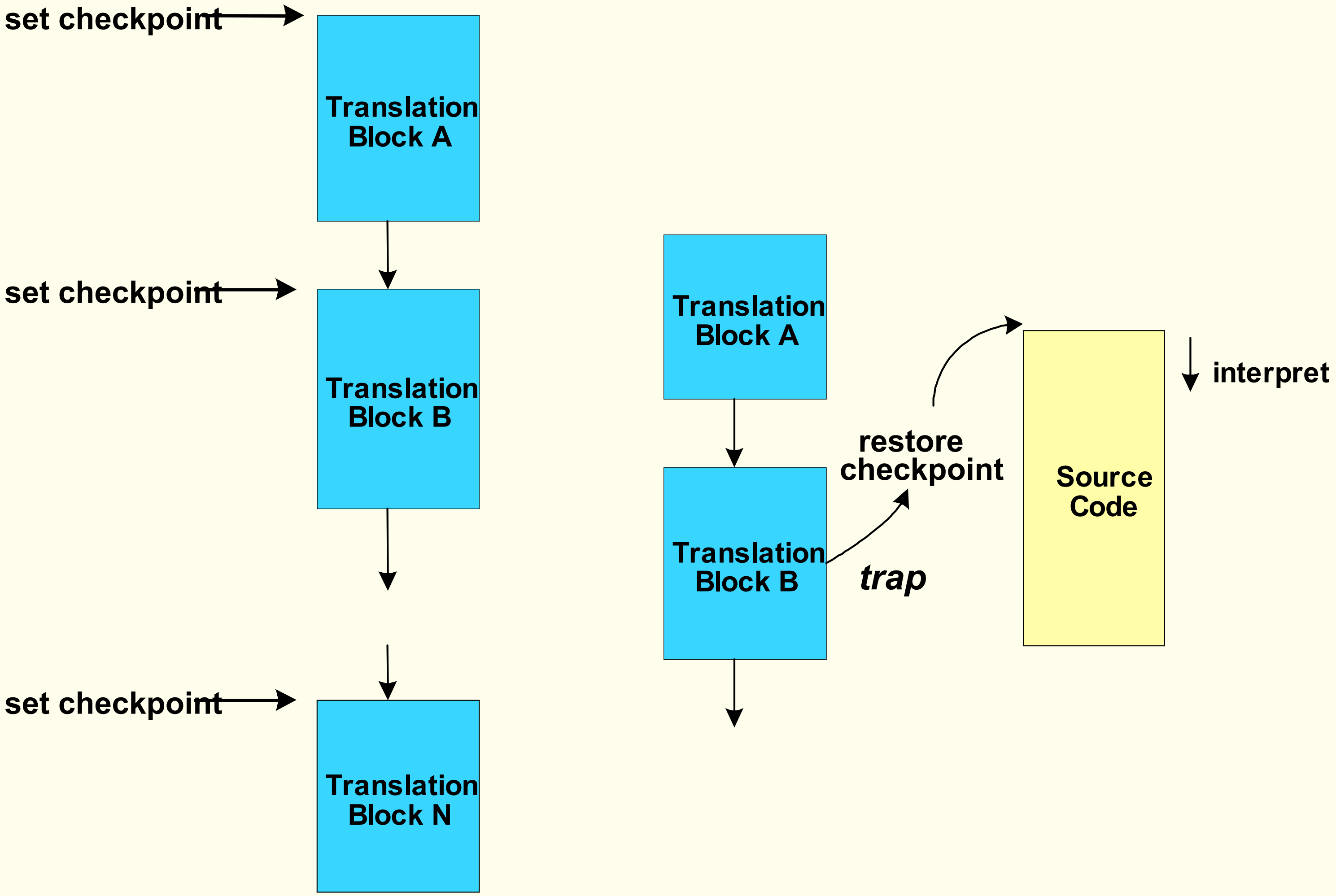
Precise Exceptions

- ❑ **Traps must be precise wrt original binary**
    - All conventional software is unaware of underlying VM
    - Code may undergo heavy duty re-organization

      E.g. CISC → VLIW
- ❑ **Checkpoint and rollback**
    - Have VMM periodically checkpoint state
    - Consistent with a point in original binary
    - On fault, rollback and interpret original binary
- ❑ **In-order state update**
    - Keep out-of-order results in scratch registers, update architected registers in-order

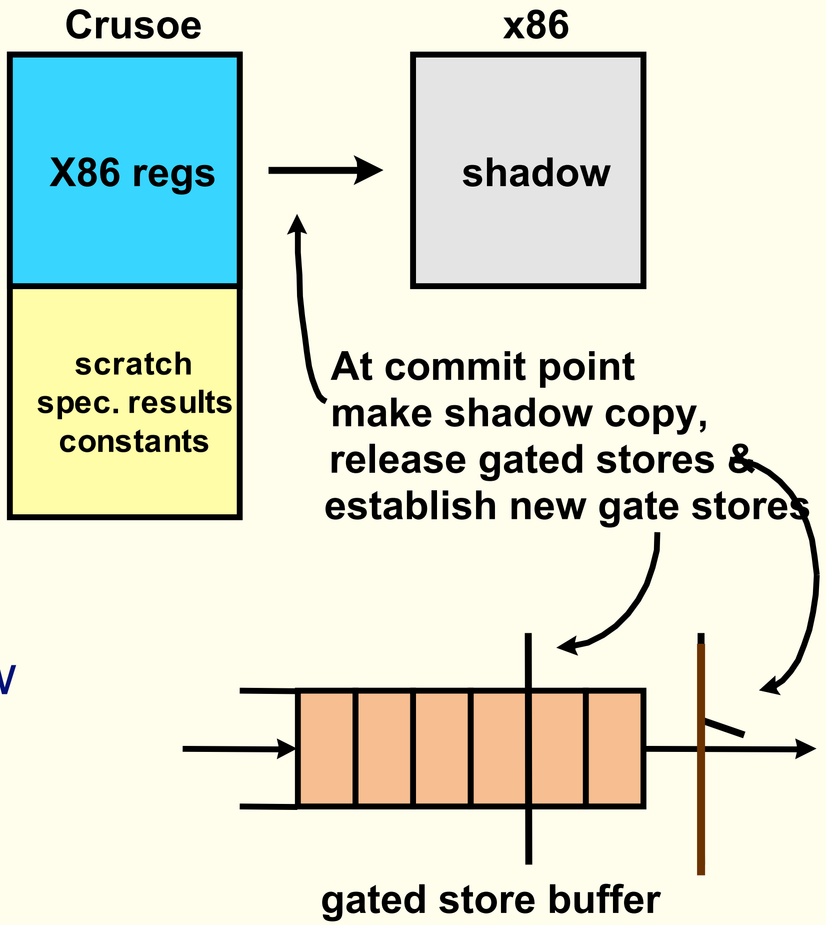
      I.e. software renaming

Checkpoint and Rollback



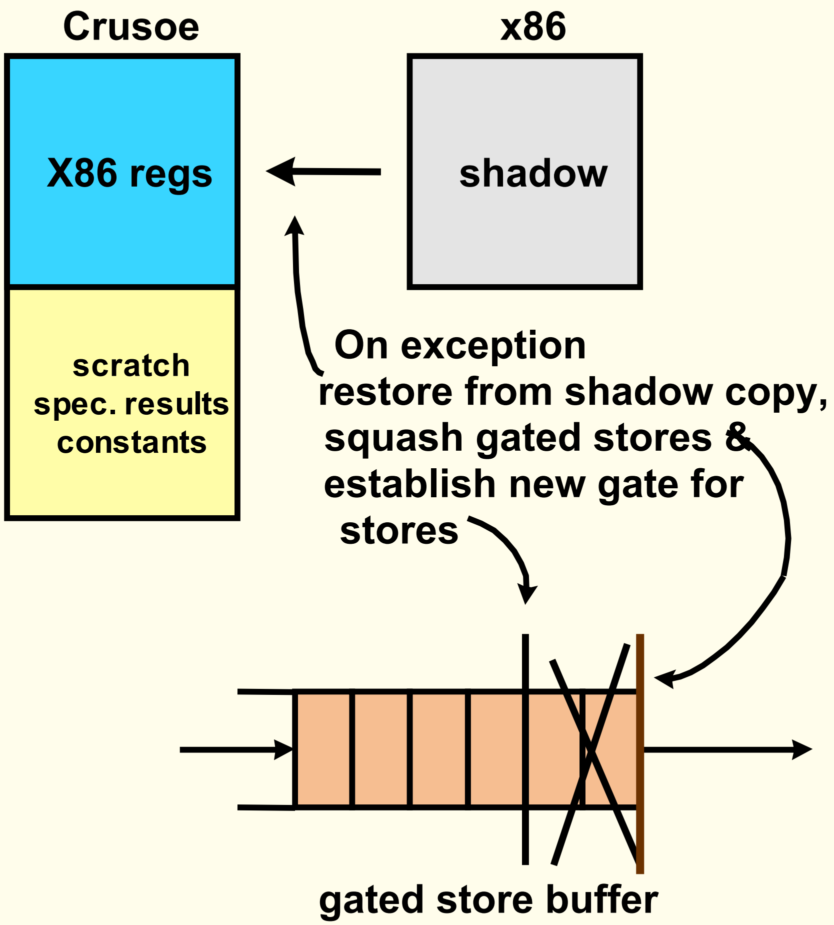
Precise Interrupts via checkpoint/rollback

- ❑ As in Transmeta Crusoe
- ❑ *Shadow* copies of registers
- ❑ Gated store buffer
- ❑ Code divided into translation groups
  - Precise state between groups
- ❑ Commit when trans. group is exited
  - Release gated store buffer
  - Copy current registers into shadow



Precise Interrupts via checkpoint/rollback

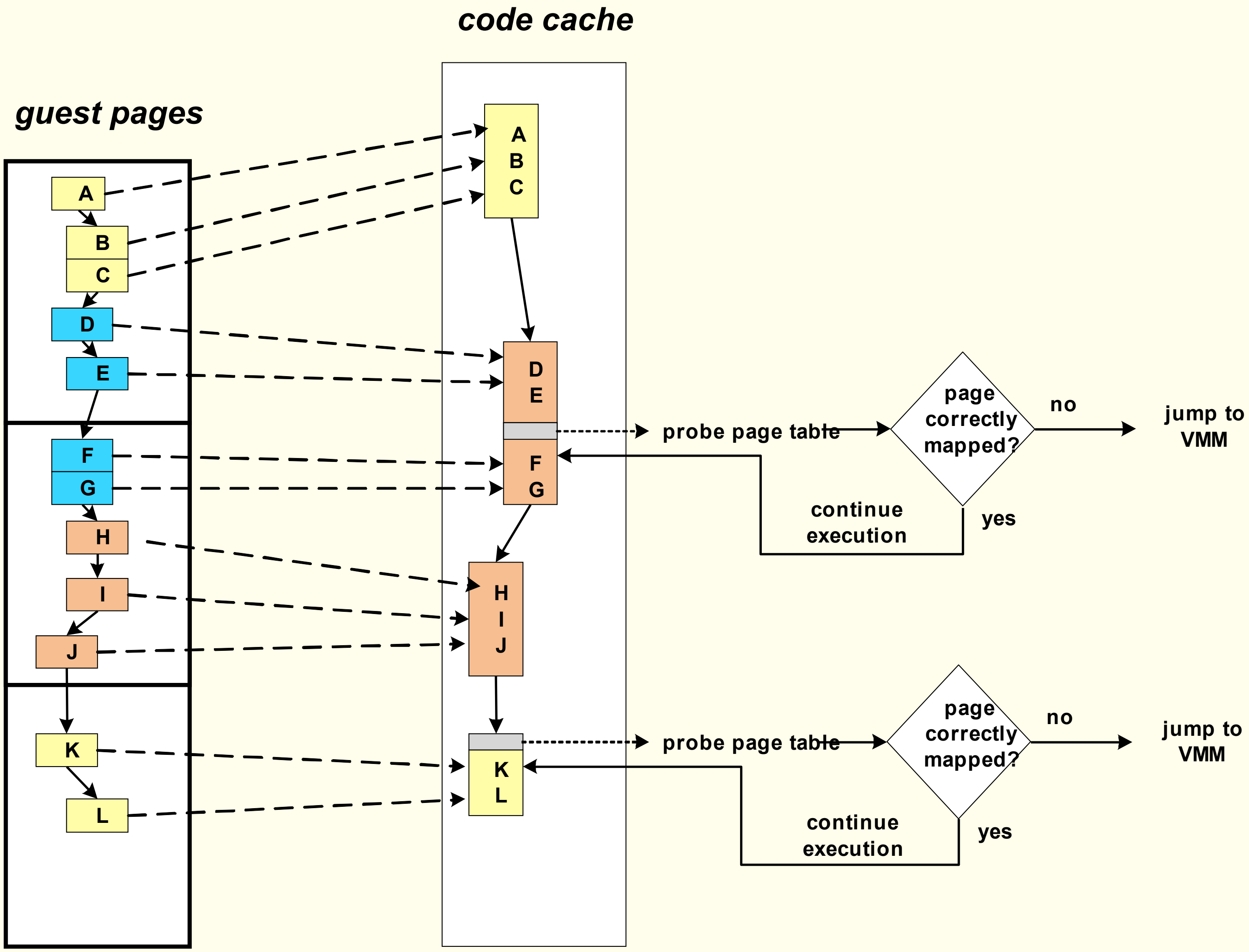
- ❑ **When a trap occurs**
  - Flush store buffer
  - Backup with shadow registers
  - Interpret forward until trap occurs
- ❑ **Advantage:**
  - Larger precise interrupt units => coarser grain optimizations, dead code elimination, etc.
- ❑ **Disadvantage:**
  - Store buffer size limits translation unit size



Page Fault Compatibility

- Major difference wrt Process VMs
- All page faults in guest must be accurately emulated
- Data accesses – no problem
  - Detected via page table/TLB
- Instruction accesses – more difficult
  - Fetches are from code cache, not guest memory
  - Code cache pages are not related to guest pages

Page Crossings

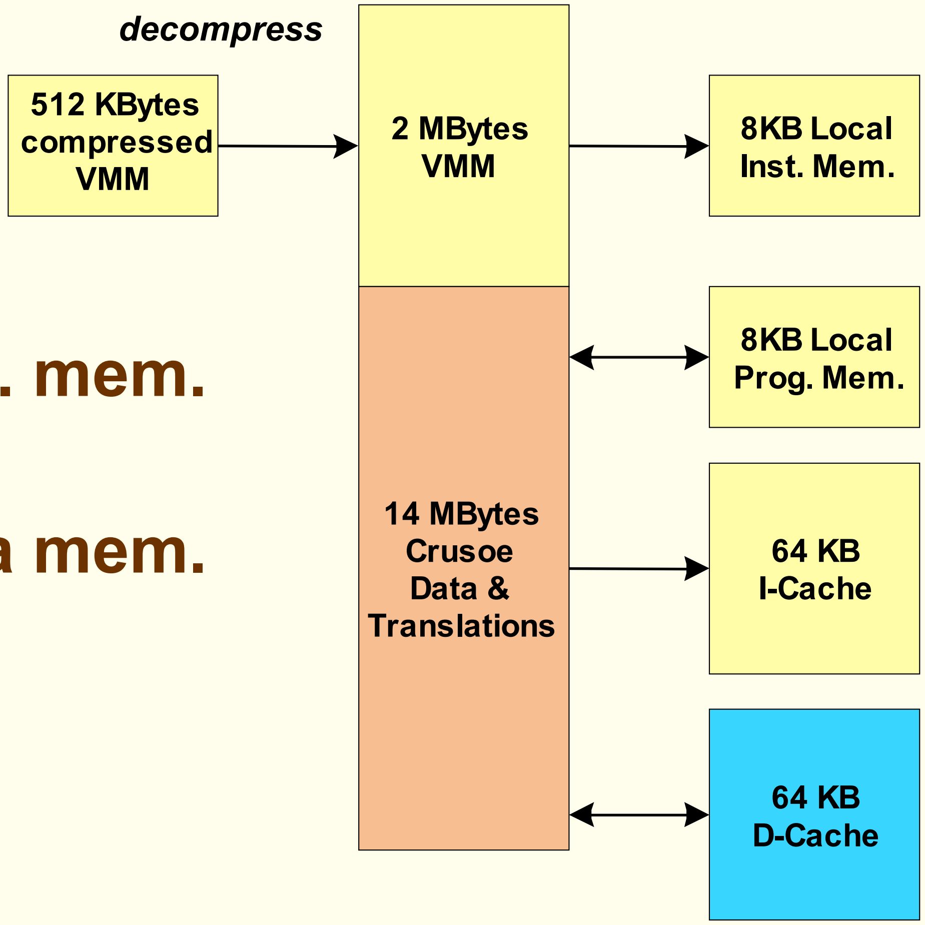


Input/Output

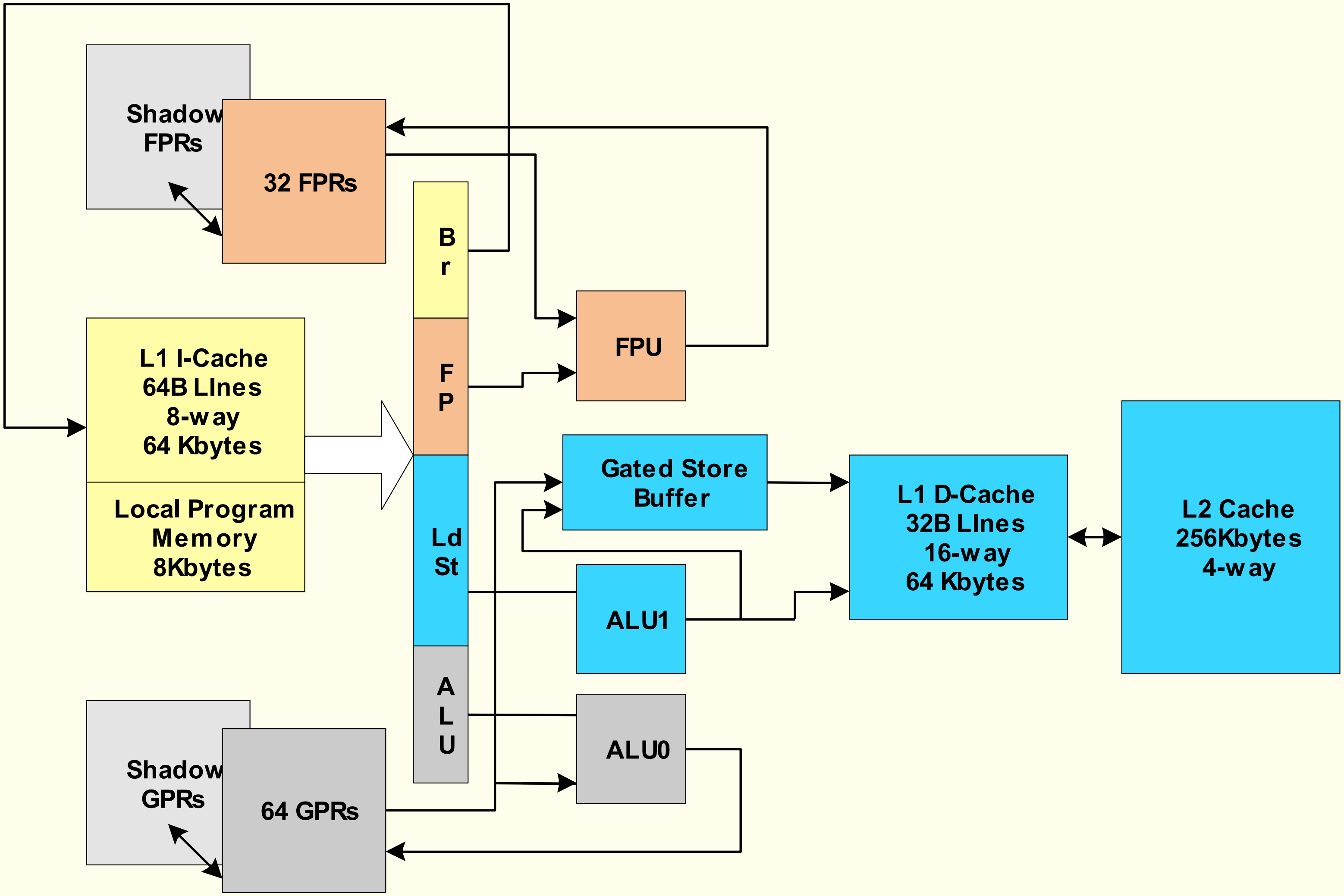
- **VMM itself uses no I/O**
- **Run guest I/O drivers as-is**
  - Let I/O drivers directly control I/O signals
- **Problems w/ Memory-Mapped I/O**
  - Use access-protect in TLB to detect accesses to volatile pages
  - De-optimize code that accesses volatile pages
  - Enhance ISA w/ load/store opcodes that over-ride access-protect

Case Study: Transmeta Crusoe

- x86 to VLIW (4-way)
  - Specialized Fields
    (ALU, LD/ST, FP, Br)
- 16M Translation Cache
- 8K bytes VMM local inst. mem.
  - Reduces I cache pollution
- 8K bytes VMM local data mem.
  - Reduces D cache pollution



Transmeta Crusoe Block Diagram



Crusoe Translation

- ❑ **Staged optimization**
  - Interpretation (with profiling)
  - Simple translation
  - Highly optimized translation
- ❑ **Algorithm translates “multiple basic blocks”**

Alias Hardware

- To allow re-scheduling of memory ops
- Removal of redundant loads
- *load-and-protect*
  - Special load opcode
  - records load address and loaded data size in table
- *store-under-alias-mask*
  - Special store opcode
  - Checks specified (via mask) loads in table
  - if conflict, triggers re-do of loads

Alias Hardware Example

```
Original Code       Optimized Code
st (W)              ldp (X)   x
. . .               . . .
ld (X)              ldp (Z)   x x
. . .               . . .
st (Y)              stam (W)  ↑
. . .               . . .
ld (Z)              stam (Y)    ↑
```

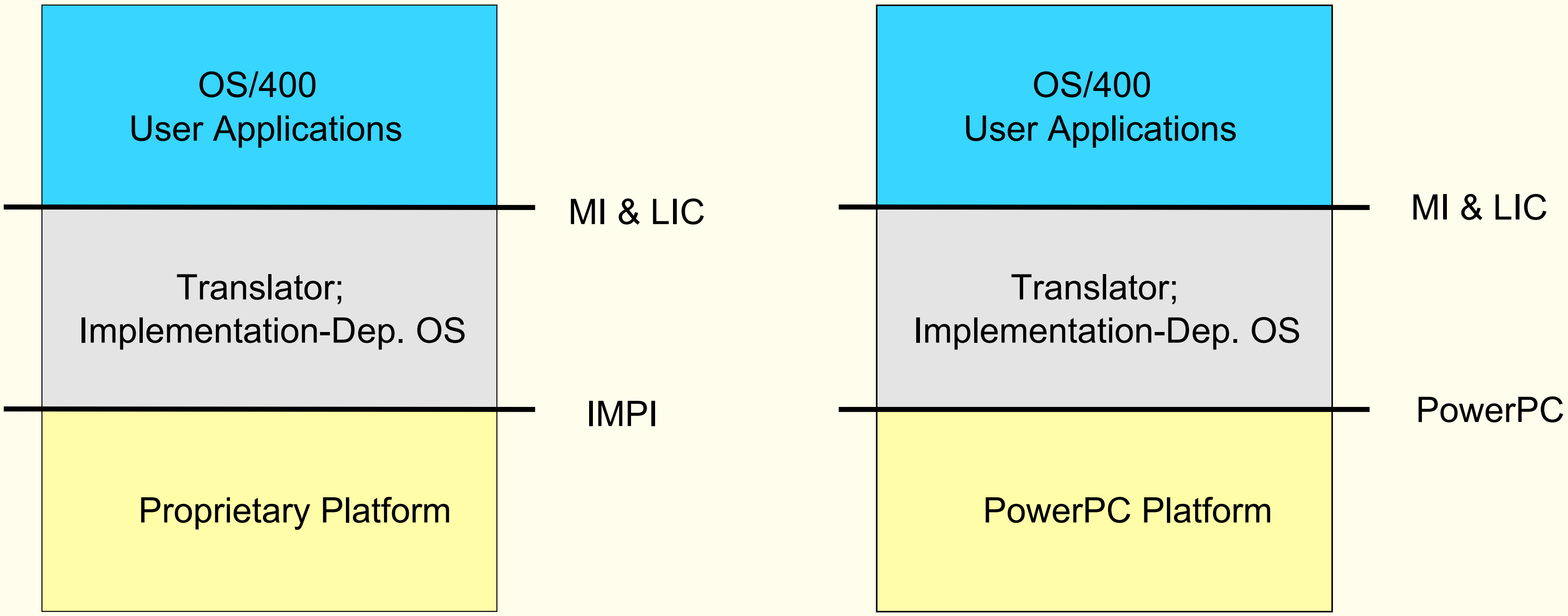
- stam(W) traps if W==X or W==Z
- stam(Y) traps if Y ==Z

Another Example: IBM AS/400

- **A very early (and successful) co-designed VM**
- **Goals**
  - Hardware independence
    - Demonstrated by move to PowerPC
  - Support robust/well integrated software
    - Re-define conventional software boundaries
    - Divided OS into implementation independent/dependent parts
    - Architect object-orientation

IBM AS/400 Platforms

- System /38 – Proprietary implementation ISA
- AS/400 – First, extend proprietary ISA
  Then transition to PowerPC ISA


OS/400
User Applications
MI & LIC
Translator;
Implementation-Dep. OS
IMPI
Proprietary Platform
OS/400
User Applications
MI & LIC
Translator;
Implementation-Dep. OS
PowerPC
PowerPC Platform