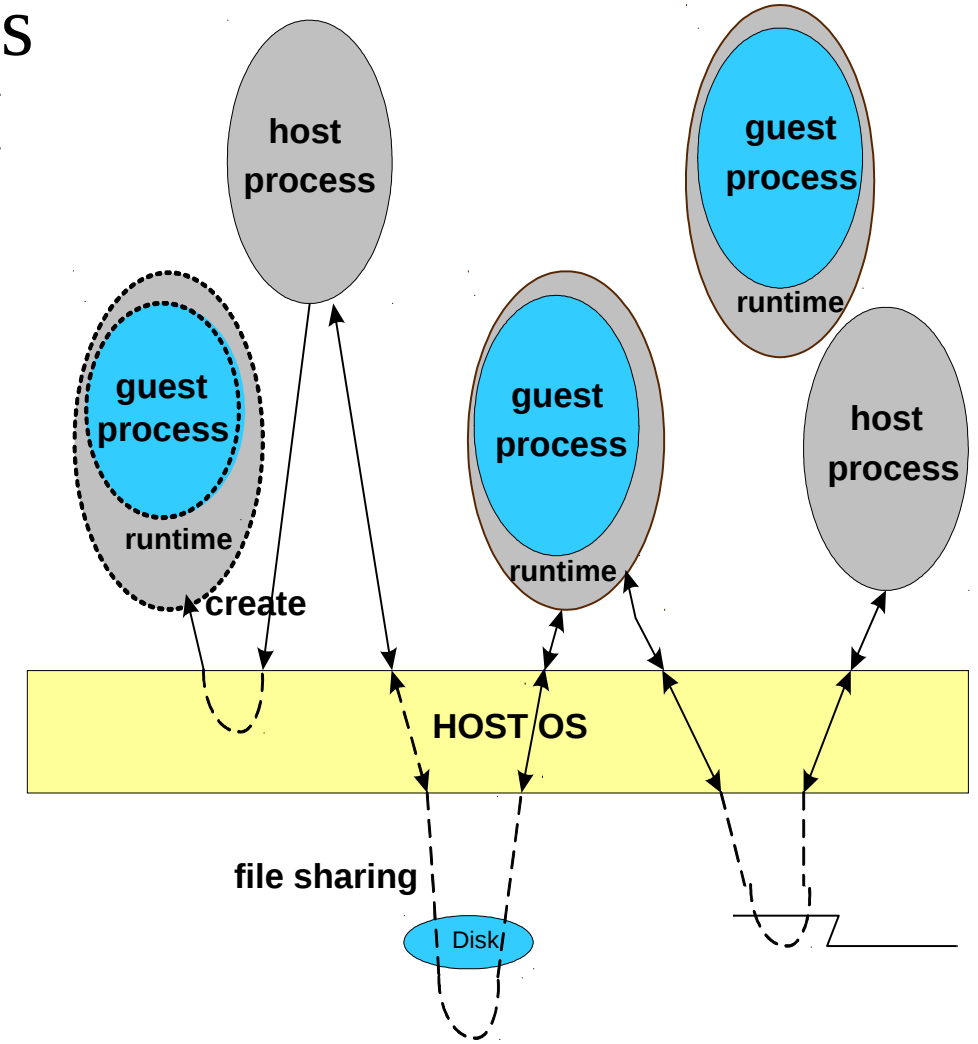
Process Virtual Machines – Outline

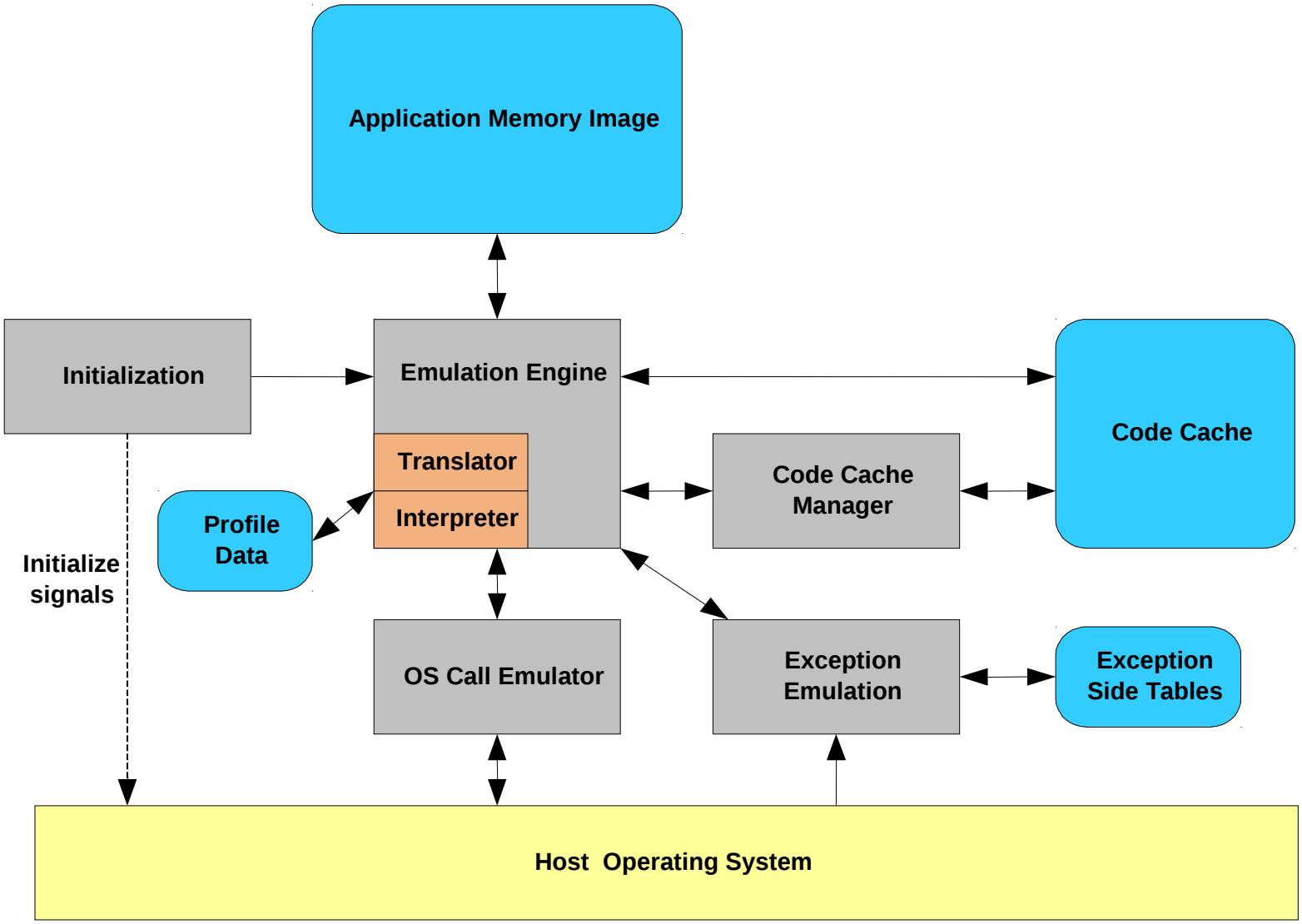
- Structure of a process VM
- Compatibility issues
- Guest-to-host state mapping issues
- Emulation of
  - memory, instructions, exceptions, and OS calls
- Profiling
- Optimization issues

Background

- Compiled applications are bound by the ABI to only work for one OS-ISA pair
  - process VMs overcome this limitation
- Example: IA-32 EL process VM with interfaces for Windows and Linux



Structure of a PVM



Structure of a PVM (2)

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  - flush outdated translations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

Compatibility

- How accurately does the emulation of the guest's *functional* behavior compare with its behavior on its native platform
  - two systems are compatible if, in response to the same sequence of input values, they give the same sequence of output values
- Intrinsic compatibility
  - precise behavior, difficult to achieve
- Extrinsic compatibility
  - accuracy within some well-defined constraints
  - acceptable for most systems

Intrinsic Compatibility

- Compatibility requires 100% accuracy for all programs all the time
  - compatible for all possible input sequences
  - no further verification needed to confirm emulation accuracy
  - difficult to achieve
- Based entirely on the properties of the VM.
- e.g., hardware designers use intrinsic compatibility to guaranty micro-architectural ISA compatibility.

Extrinsic Compatibility

- Compatible for well-defined subset of input sequences
  - based on VM implementation, architecture/OS specifications, and external guarantees or certificates
  - some burden on the users to ensure that guarantees are met
- e.g., VM may only guaranty accuracy for programs compiled with a particular compiler
- e.g., program may be compatible as long as it has limited resource requirements

Verifying Compatibility

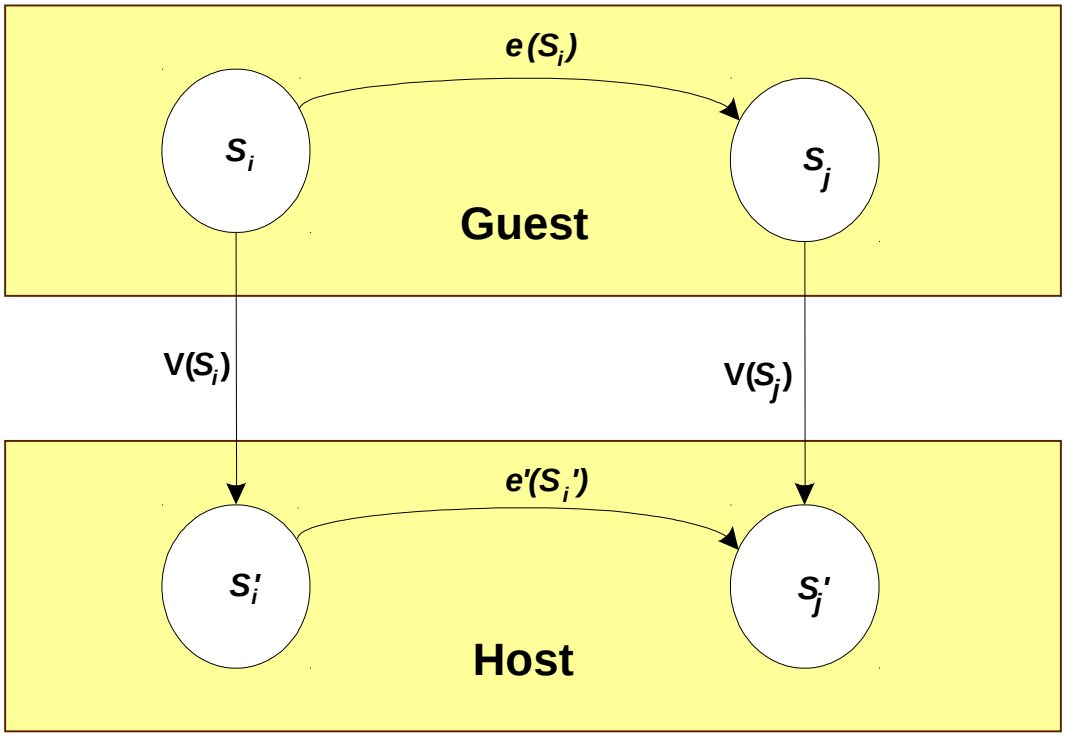
- Too complex to theoretically prove
  - except in simple systems
- In practice
  - use informal reasoning
  - use test suites
- Sufficient conditions
  - decompose compatibility into parts
  - allows the reasoning process to be simplified
- Assume state of guest is 1 to 1 mapped to host
  - but same “type” of state is not necessary

A Compatibility Framework

- The need for a framework
  - rigorously proving that compatibility holds is hard
  - allow to reason about compatibility issues
  - decide when/where during program execution should compatibility be guaranteed/verified
- Model of program execution
  - machine *state*, defined by registers, memory, I/O, etc.
  - *operations* that change state

A Compatibility Framework (2)

- Guaranty isomorphic mapping between guest and host states

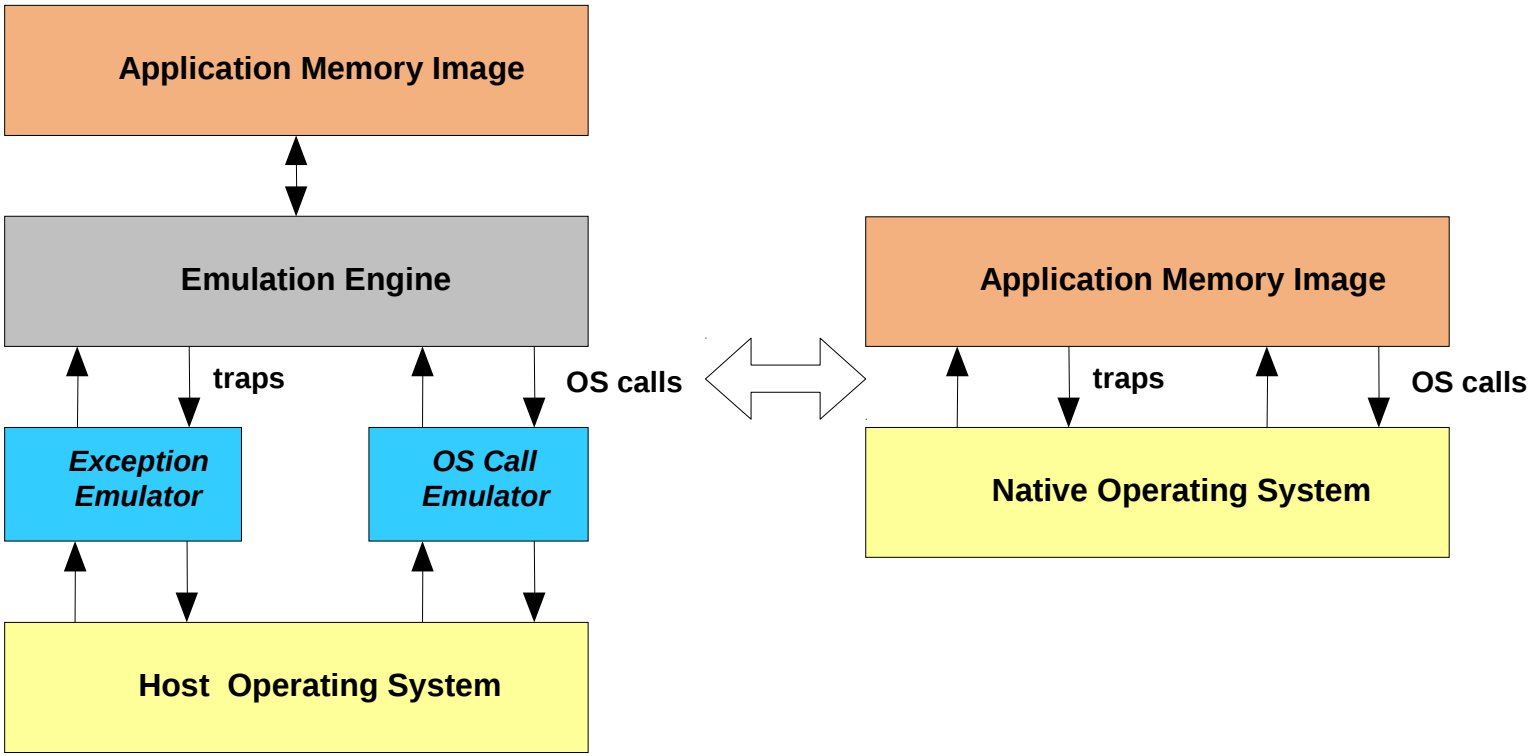


Compatibility Framework (3)

- Managing (changes to) program state at two levels
  - user-managed state
    - main memory, registers
    - straightforward mapping between guest and host states
    - operated on by user-level instructions
  - OS-managed state
    - disk contents, I/O state, networks
    - operated via OS calls, traps, interrupts
    - operations can affect user-level state as well

Compatibility Framework (4)

- Compatibility is only verified at points where control is transferred between the user code and OS
  - establish one-to-one mapping between control transfer points in both native platform and VM

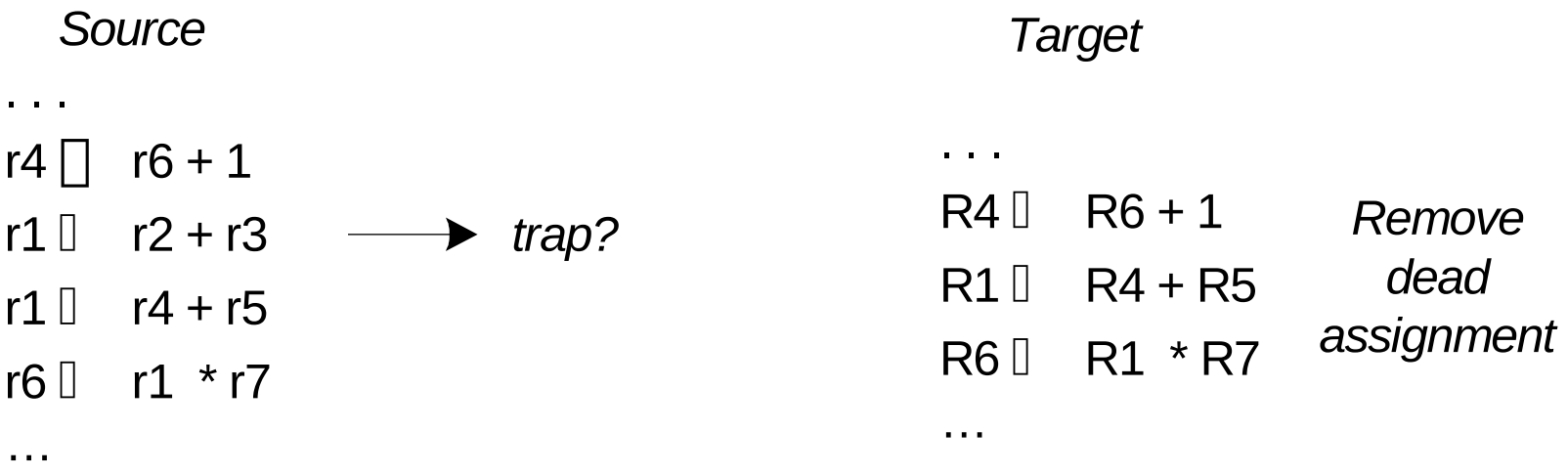


Compatibility Framework (5)

- Conditions for compatibility
  - guest state should be equivalent to host state at
    - control transfer from user instructions to OS
    - control transfer from OS to user instructions
  - *all* user-managed state must be compatible
  - instruction-level equivalence not required

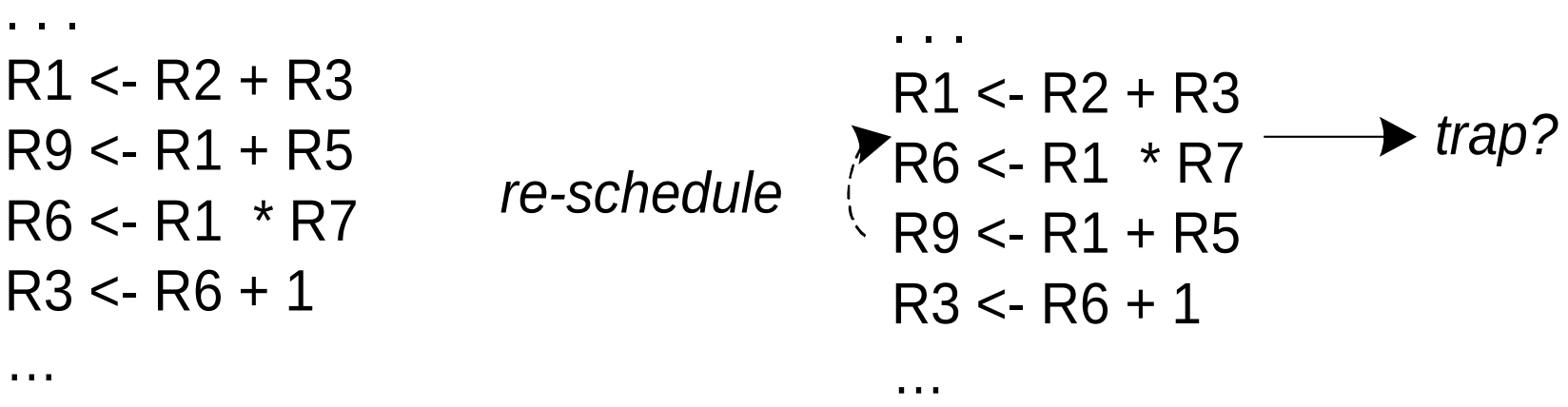
Trap Compatibility

- If source traps, then target traps
- If target traps, then source would have trapped
  - runtime can filter target traps, to remove false ones
- Page faults are special case
  - page fault behavior is non-deterministic w.r.t. user process



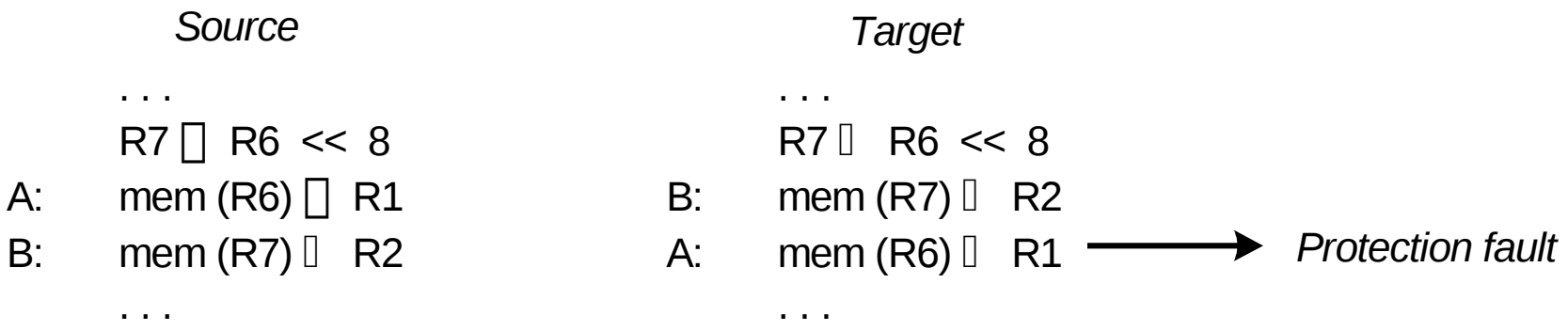
Register State Compatibility

- At the time of an exception is the register state exactly as in the real machine?
  - including dead register values?



Memory State Compatibility

- *Memory state compatibility* is maintained if, at the time of a trap or interrupt, the contents of memory are exactly the same in the translated target program as in the original source program.



Memory Ordering Compatibility

- Maintain equivalent consistency model
- Important for multiprocessors

```
A = Flag = 0;
```

Process P1

```
A = 1;
Flag = 1;
```

Process P2

```
while (Flag == 0);
.... = A;
```

Undefined Architecture Cases

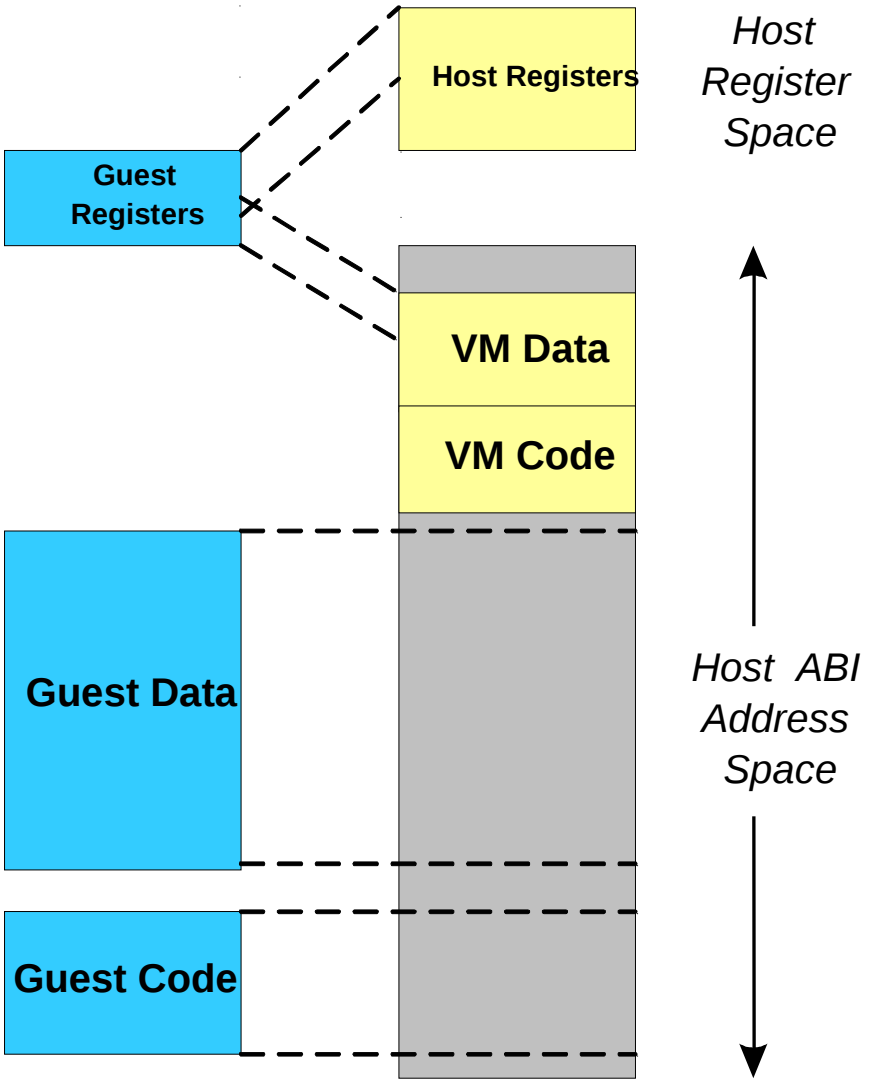
- Some (most?) ISAs have undefined cases
  - example: self-modifying code with I-caches
  - unless special actions are performed, result may be undefined
- Different, undefined behavior is compatible behavior
  - can be tricky – what if undefined behavior is different from all existing implementations?
  - what if existing implementations do the “logical” thing?
    - e.g., self-modifying code works as “expected”

Constructing a Process VM

- Mapping of user-managed state
  - held in registers
  - held in memory
- Perform emulation (operations to transform state)
  - memory architecture emulation
  - instruction emulation
  - exception emulation
  - OS emulation

State Mapping

- Map user-managed register & memory state
  - guest data and code map into host's address space
  - host address space includes runtime data and code
  - guest state does not have to be maintained in the same type of resource
- Register mapping
  - straight-forward
  - depends on number of guest and host registers

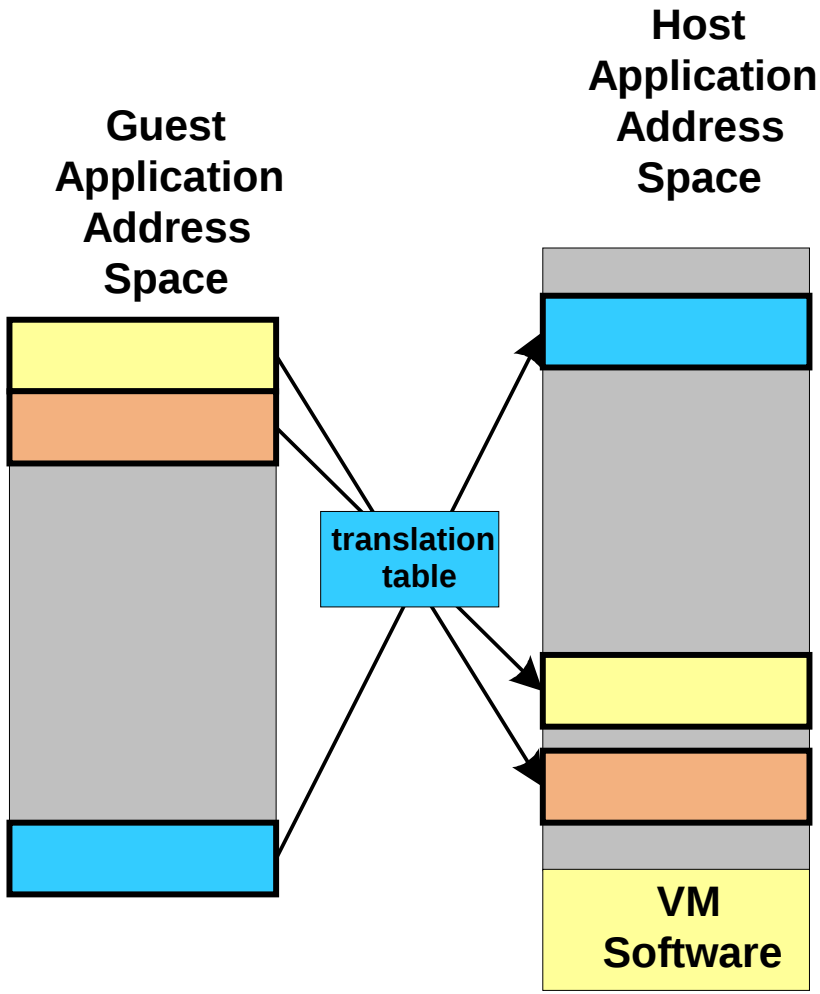


Memory State Mapping

- Memory address space mapping
  - map guest address space to host address space
  - maintain protection requirements
- Methods – results in different performance and flexibility levels
  - software supported translation table
  - direct translation

Software Translation Tables

- VM software maintains *translation table*
  - map each guest memory address to host address
  - similar to hardware page tables / TLBs
  - used when all other approaches fail
  - provides most flexibility and least performance

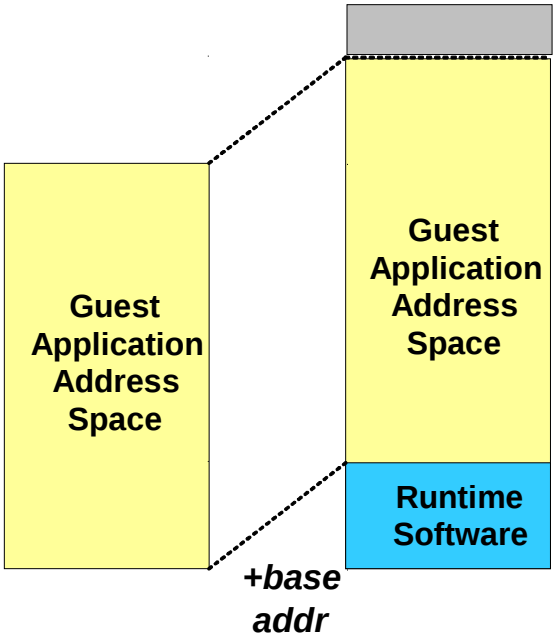


Software Translation Tables (2)

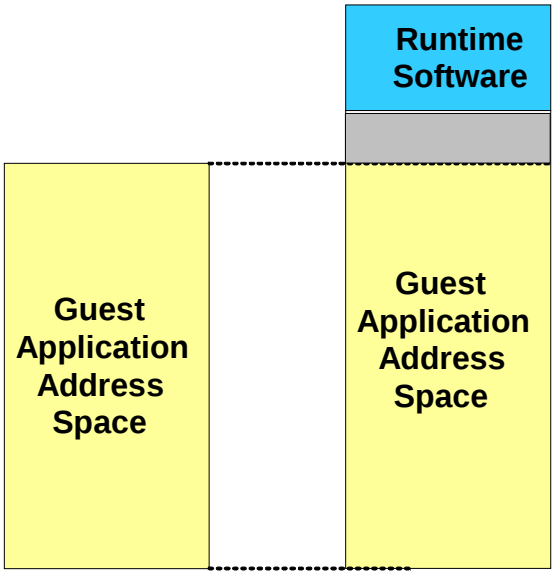
```
Initially,  R1 holds source address
            R30 holds base address of mapping table


srwi   r29,r1,16    ;shift r1 right by 16
slwi   r29,r29,2    ;convert to a byte address
lwzx   r29,r29,r30  ;load block location in host memory
slwi   r28,r1,16    ;shift left/right to zero out
srwi   r28,r28,16   ;source block number
slwi   r29,r29,16   ;shift up target block number
or     r29,r28,r29  ;form address
lwz    r2,0(r29)    ;do load
```

Direct Memory Translation

- Use underlying hardware
  - guest memory allocated contiguous host space
  - guest address space + runtime <= host address space
  - minimal overhead, most performance



fixed non-zero offset



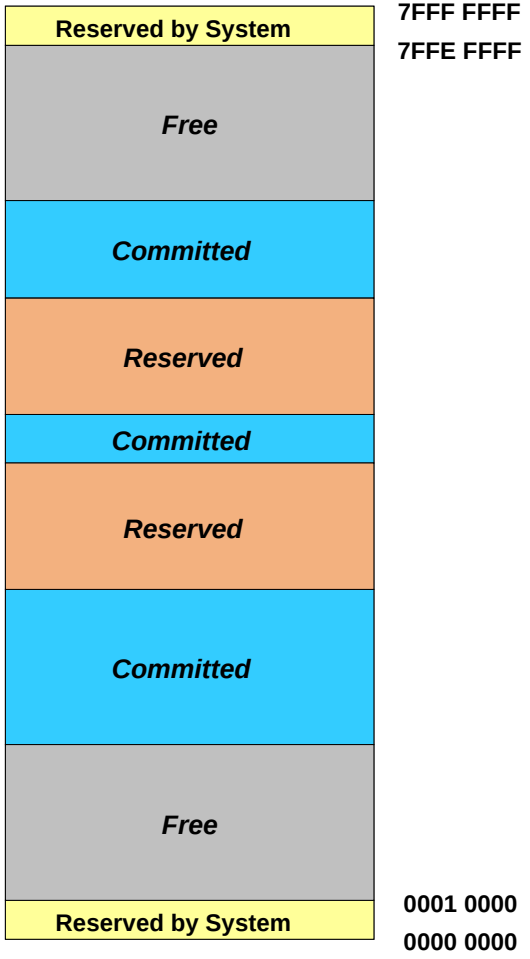
zero offset

Memory State Mapping – Summary

- Runtime + guest space <= host space
  - direct memory translation
  - can achieve performance and intrinsic compatibility
- Runtime + guest space > host space
  - software translation
  - will lose intrinsic compatibility, performance or both
- guest space == host space
  - happens often, same-ISA dynamic translation
  - no room for runtime
    - use software translation, extrinsic compatibility

Memory Architecture Emulation

- Aspects of the ABI memory architecture that need to be emulated.
- Address space structure
  - segmented or flat
- Access privilege types
  - combination of N, R, W, E
- Protection / allocation granularity
  - size of the smalled block of memory that can be allocated by the OS

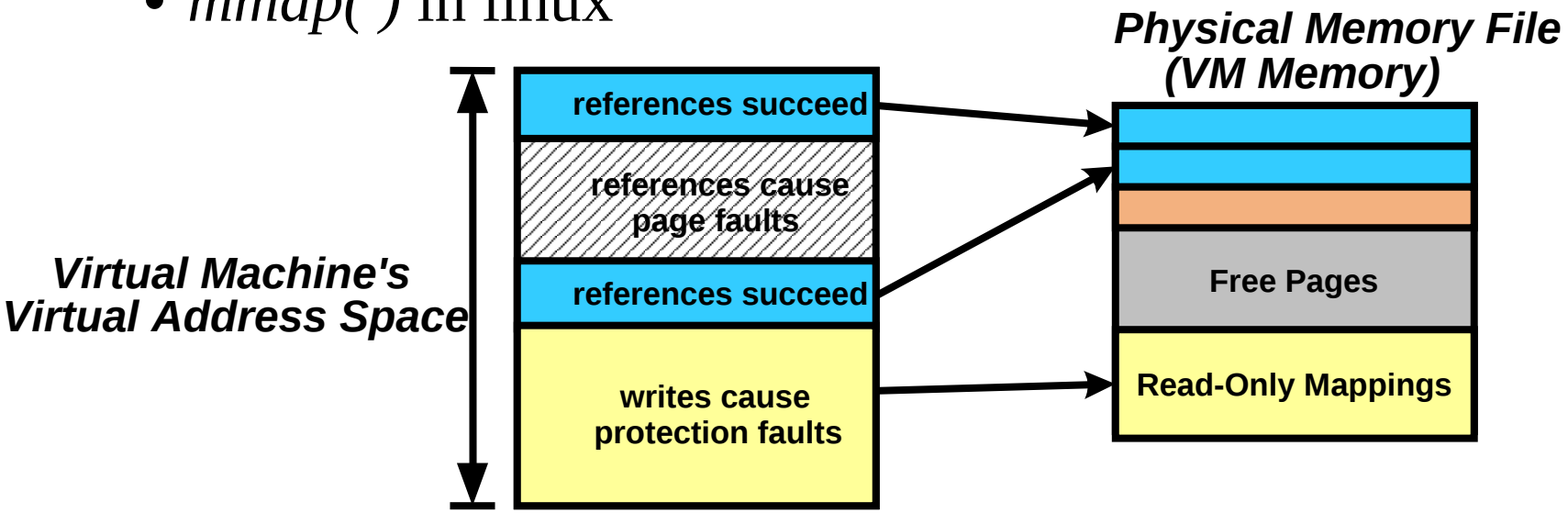


Guest Memory Protection

- Access restrictions placed on different regions of memory.
- Can be achieved during software supported translation
  - slow and inefficient, but very flexible
- Host supported memory protection
  - runtime sets access restrictions using OS system calls
  - OS delivers signals to runtime on access violations
  - protection faults reported to runtime
  - requires host OS support

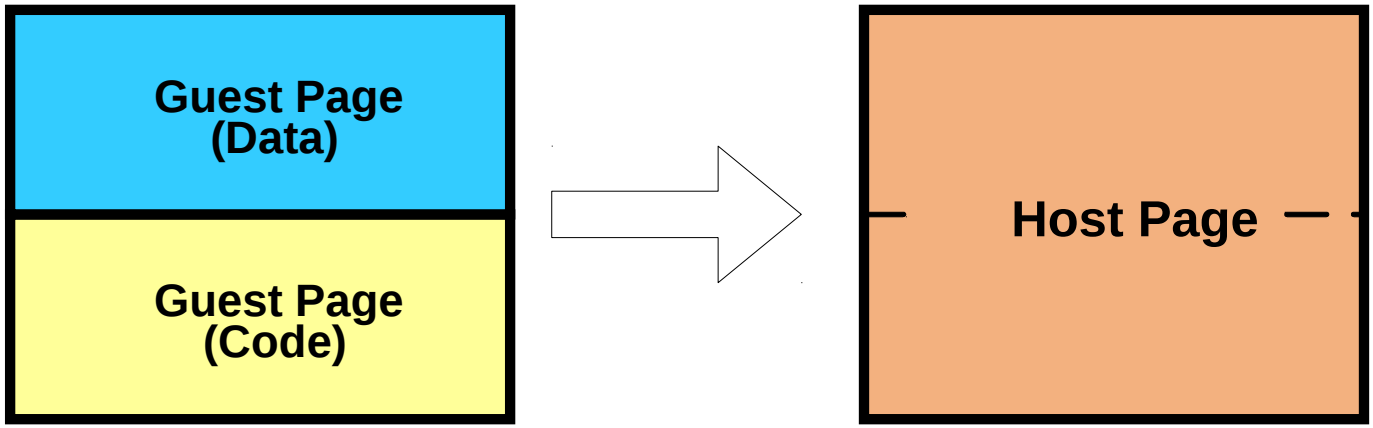
Host OS Support

- Direct mechnism
  - runtime sets protection levels via system calls (*mprotect*)
  - protection faults trap to handler in runtime (*SIGSEGV*)
- Indirect mechanism
  - mapping region of memory to file with access protections
    - *mmap( )* in linux



Guest Memory Protection (2)

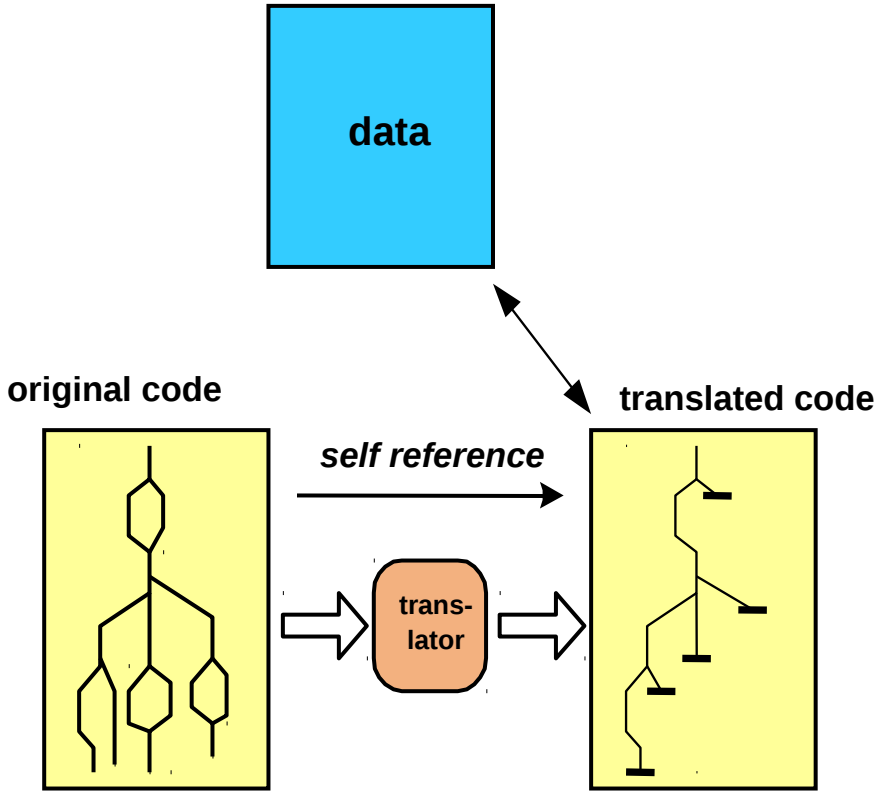
- Implementation issues
  - host and guest ISAs provide different protection types
    - host provides a superset of guest protections
    - host provides a subset of guest protections
  - host and guest support different page sizes
    - difficult to map access privileges
    - simple if guest page size is a multiple of host page size



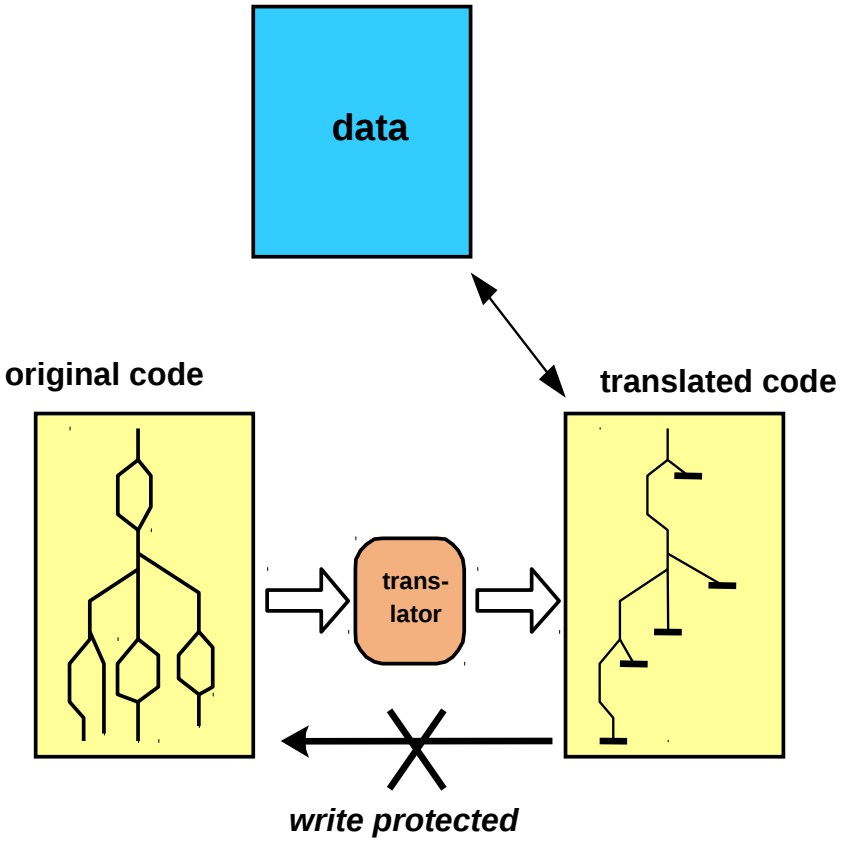
Self-Referencing/Modifying Code

- Program may either refer to itself, or attempt to modify itself.
- Solution
  - maintain guest program code memory image
  - load/store addresses are mapped into source memory region
  - loads from code region are ok
  - writes to code region trigger segfault
    - flush relevant cache entry, enable writes to code region, interpret the code block that caused the fault, re-enable write-protection

Self-Referencing/Modifying Code (2)



Self – referencing code



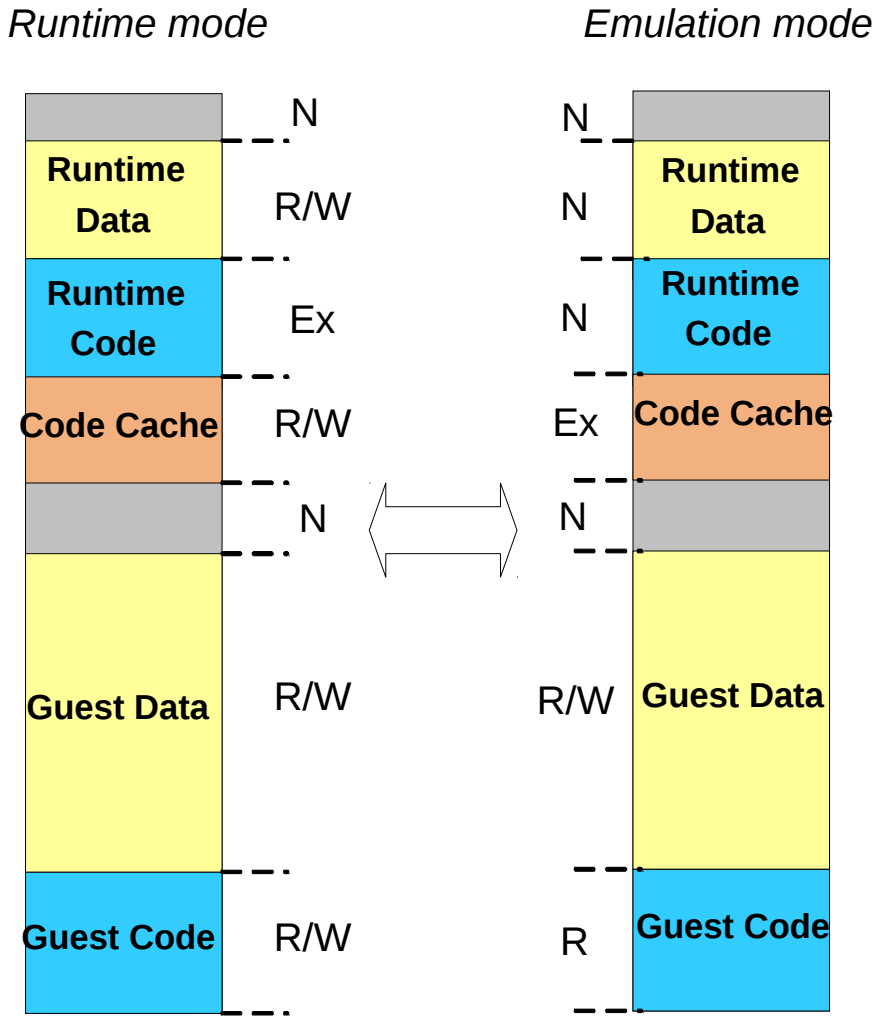
Self – modifying code

Protecting Runtime Memory

- Runtime and guest application share the same process address space
  - guest program can read/write portions of the runtime
- Addressing
  - software translation tables
  - hardware address translation, software protection checking
  - hardware for both address translation and protection checking
    - OS sets protections for *emulation* mode and *runtime* mode
    - see Figure 3.16

Protecting Runtime Memory (2)

- Change protections on *context switch* from runtime to translated code
- Translated code can only access guest memory image
- Translated code cannot jump outside code cache (emulation s/w sets up links)
- Multiple system calls at context switch time
  - high overhead

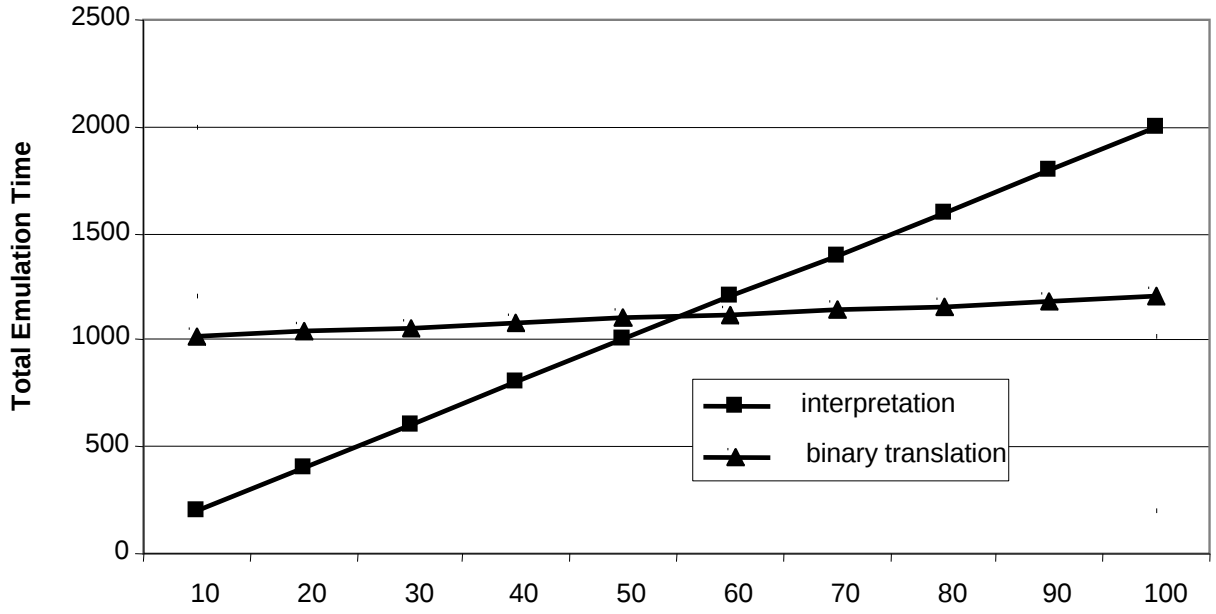


Instruction Emulation

- Techniques for instruction emulation
  - interpretation, binary translation
- Start-up time (S)
  - cost of translating code for emulation
  - one time cost for translating code
- Steady-state performance (T)
  - cost of emulation
  - average rate at which instructions are emulated

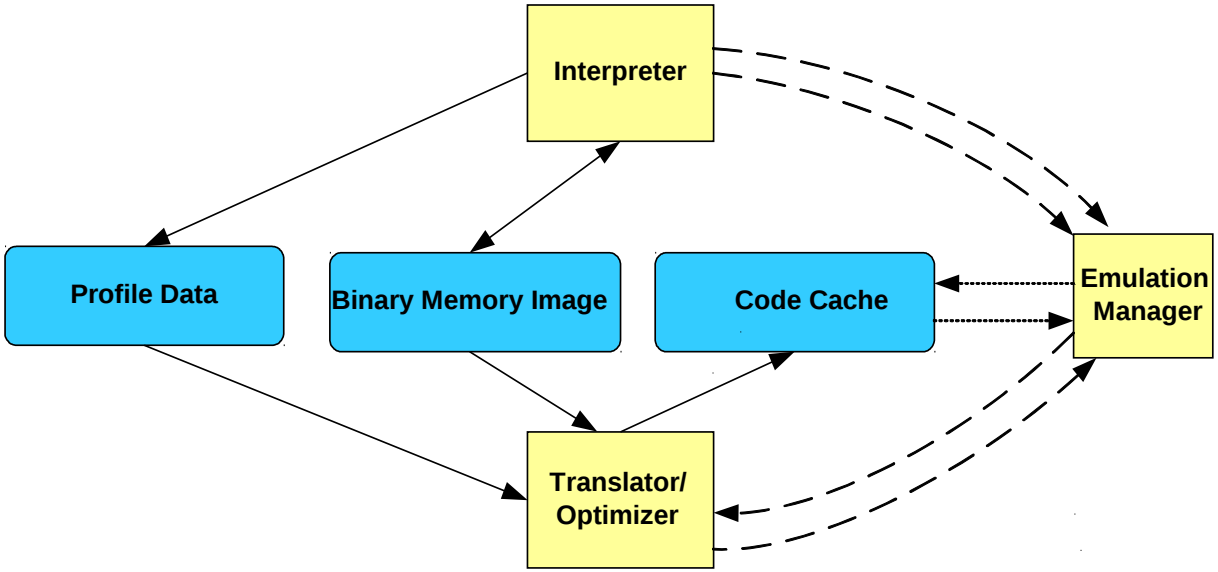
Instruction Emulation (2)

- Overall performance (S + NT)
  - N is the number of times an instruction is executed
  - S=1000, T=2/20, tradeoff point=55ins

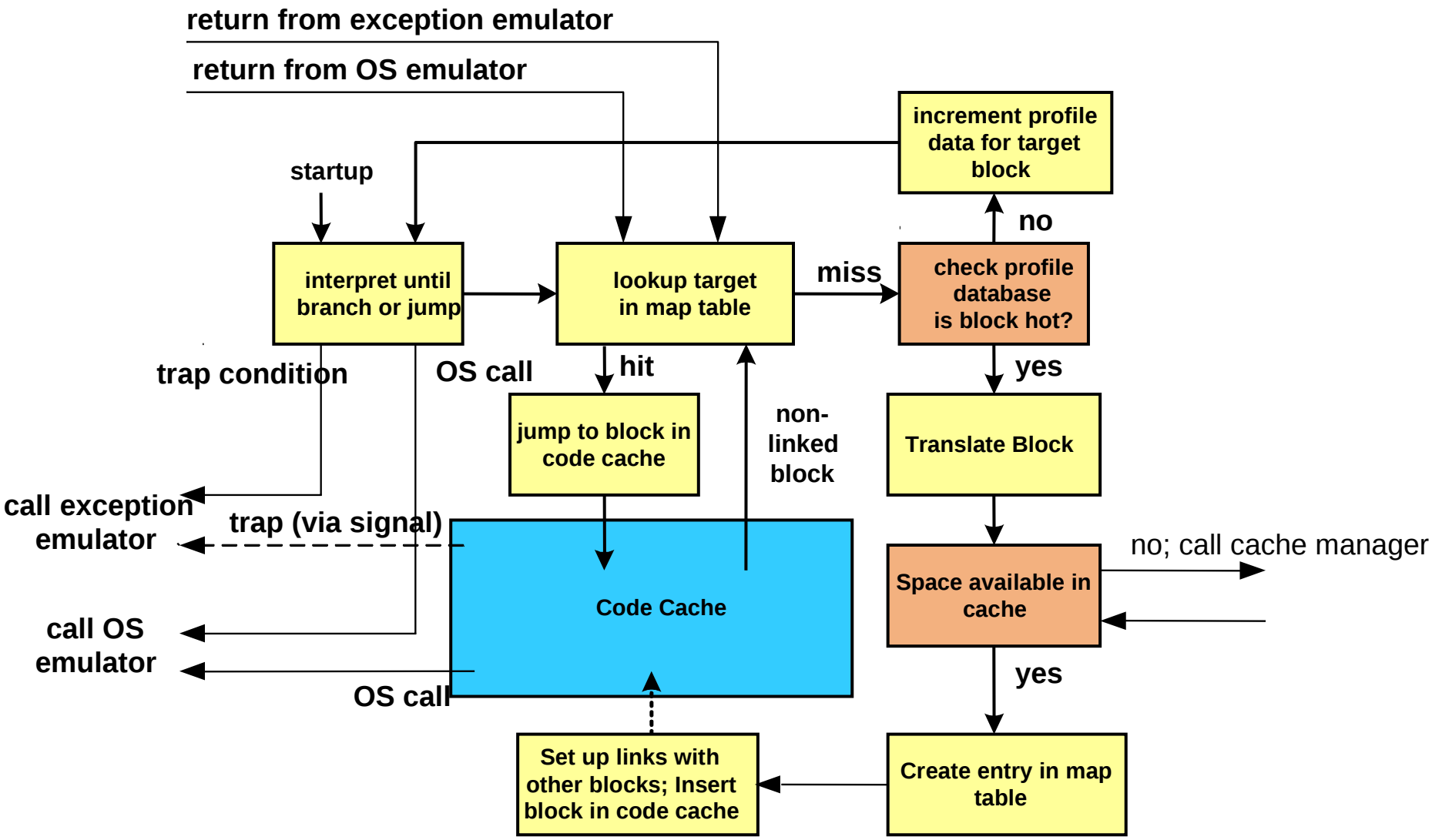


Staged Emulation

- Application of emulation techniques in stages
  - start with low start-up overhead tech. (interpretation)
  - profile data determines hot dynamic blocks of code
  - if execution count > threshold, then compile
  - place in code cache, update links and side table entries
  - optimize *hotter* code further ?



Emulation Engine Execution Flow



Exception Emulation

- Types of exceptions
  - *trap*: produced by a specific program instruction during program execution
  - *interrupt*: an external event, not associated with a particular instruction
- Precise exceptions
  - all prior instructions have committed
  - none of the following instructions have committed
- Further division of exceptions for a process VM
  - ABI visible: exceptions returned to the application via an OS signal
  - ABI invisible: ABI is unaware of the exception's occurrence

Trap Detection

- Detecting trap conditions
  - *interpretive trap detection*: checking trap conditions during interpretation routine
  - trap condition detected by the host OS
- Implementation
  - runtime registers all exceptions with the host OS
  - all signals registered by the guest program are recorded
  - on receiving OS signal, if signal is guest-registered then send to guest signal-handling code
  - else, runtime handles the trap condition
  - special tables needed during binary translation

Interrupt Handling

- Interrupts are not associated with any instruction
  - a small response latency is acceptable
  - maintaining precise state easier than traps
- Receiving interrupt during interpretation
  - complete current routine
  - service interrupt
- Receiving interrupt during binary translation
  - execution may not be at an interruptible point
  - precise recovery at arbitrary points difficult
  - no idea when control will return to the EM from the code cache

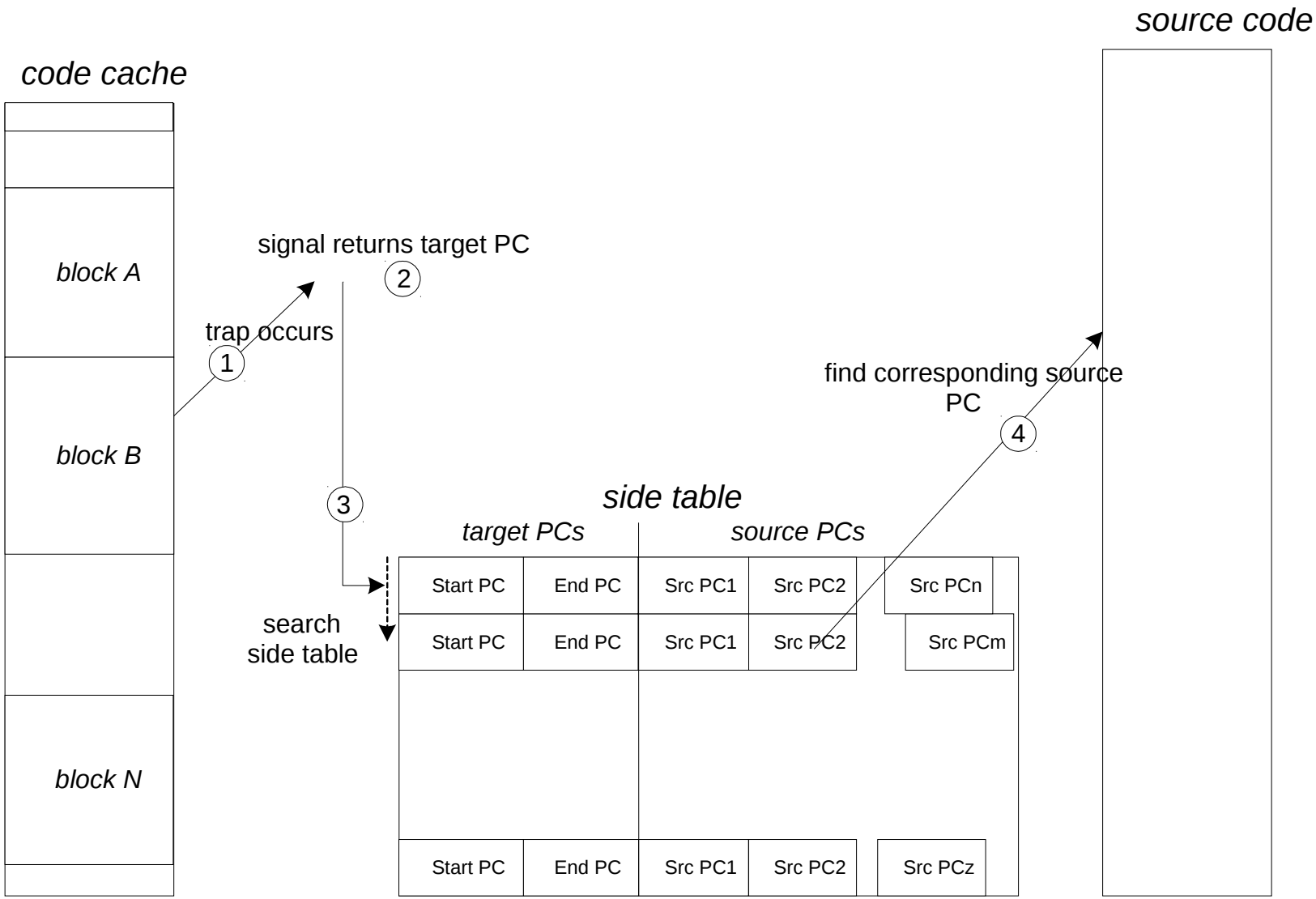
Interrupt Handling (cont…)

- Solving the interrupt response time problem during binary translation
  - on interrupt, control is passed to runtime
  - runtime unlinks the current translation block from the next block
  - control is returned back to translated code
  - control returns to runtime after end of current block
  - runtime handles the interrupt

Determining Precise State

- Interpreter
  - easy, each source instruction has its own routine
  - source PC and state updated in each instruction routine
- Binary Translation
  - hard, first determine the source PC
  - source PC not continuously updated
  - maintain *reverse translation table* mapping target PC to source PC, inefficient
  - target instruction can map to multiple source instructions
  - target code may be optimized, and re-ordered

Reverse Translation Table



Restoring Precise State

- Register state (during binary translation)
  - 2 cases, based on if source-to-target register mapping remains constant throughout emulation
  - if not constant, side tables can be maintained, or analyze from start of translation block again
- Memory State (during binary translation)
  - changed by store instructions
  - do not reorder stores, or other potentially trapping instructions with stores
  - restricts optimizations

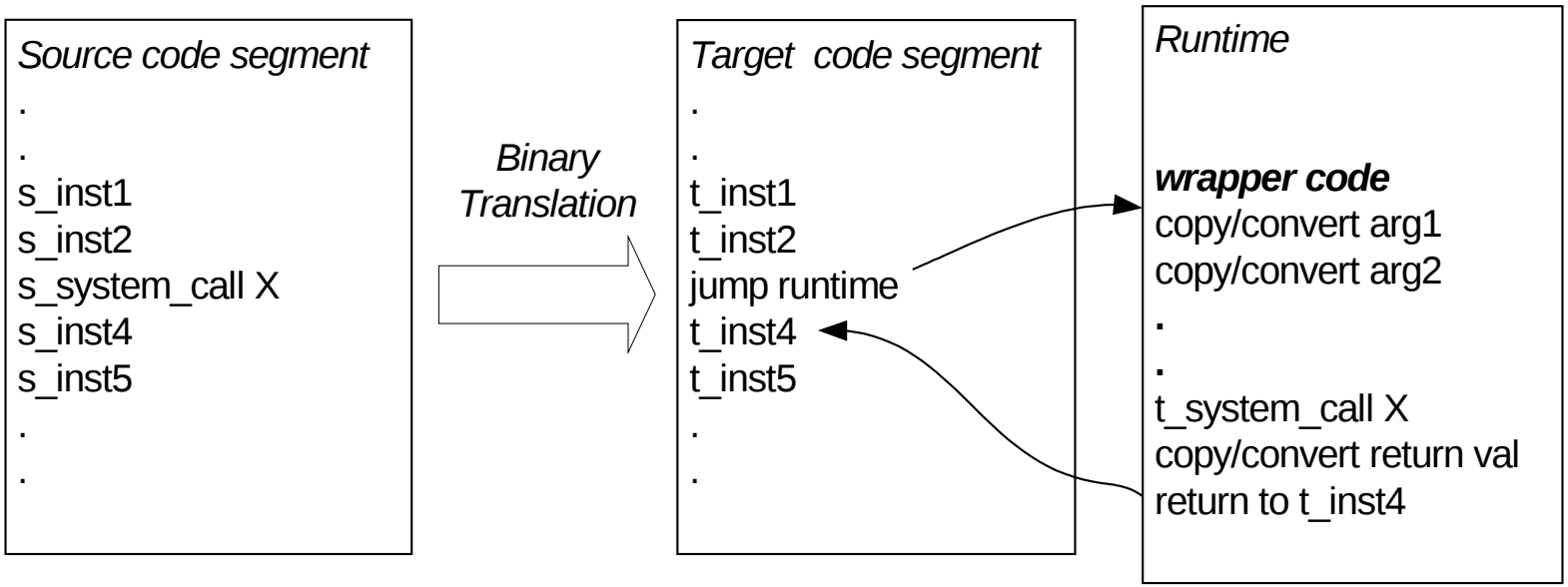
OS Call Emulation

- A PVM emulates the function or semantics of the guest's OS calls
  - not emulate individual instructions in the guest OS
- Different from instruction emulation
  - given enough time, any function can be performed on the input operands to produce a result
  - most ISAs perform same functions, ISA emulation is always possible
  - with OS, it is possible that providing some host function is impossible, operation semantic mismatch

OS Call Emulation (2)

- Different source and target OS
  - semantic translation of mapping required
  - may be difficult or impossible
  - ad-hoc process on a case-by-case basis
- Same source and target OS
  - emulate the guest calling convention
  - guest system call jumps to runtime, which provides wrapper code

OS Call Emulation (3)



- Same source and target OS (cont...)
  - runtime may handle some guest OS calls itself (signals, memory management)
  - handling abnormal conditions like callbacks, runtime maintaining program control, lack of documentation

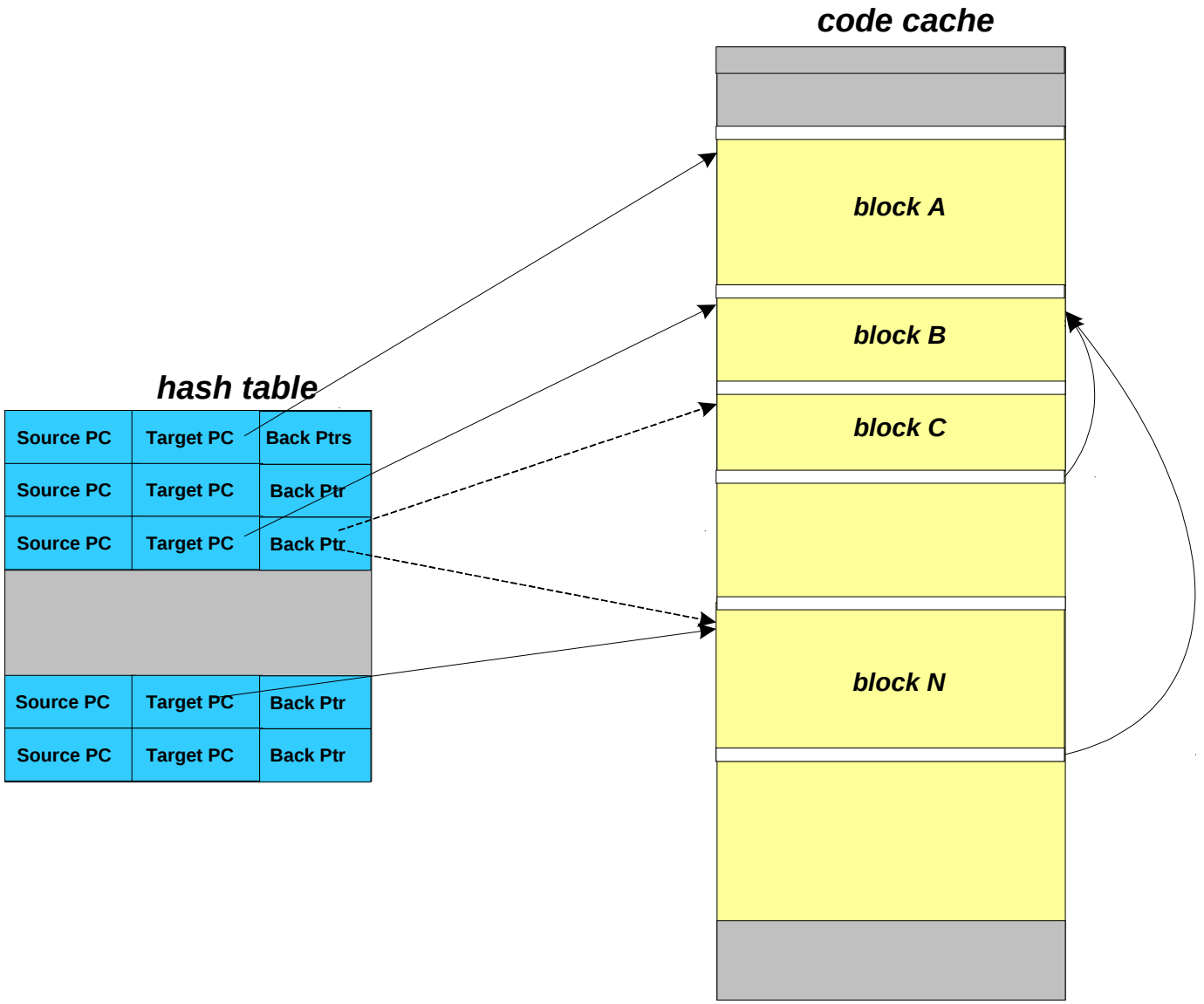
Code Cache

- Storage space for holding translated guest code.
- Code cache is different from ordinary caches
  - code cache blocks do not have a fixed size
  - code cache blocks are chained with each other
  - code cache blocks are not *backed up*
  - has implications on code cache management (replacement) algorithms used
- Code cache space is limited
  - blocks need to be replaced if cache fills up

Code Cache Replacement

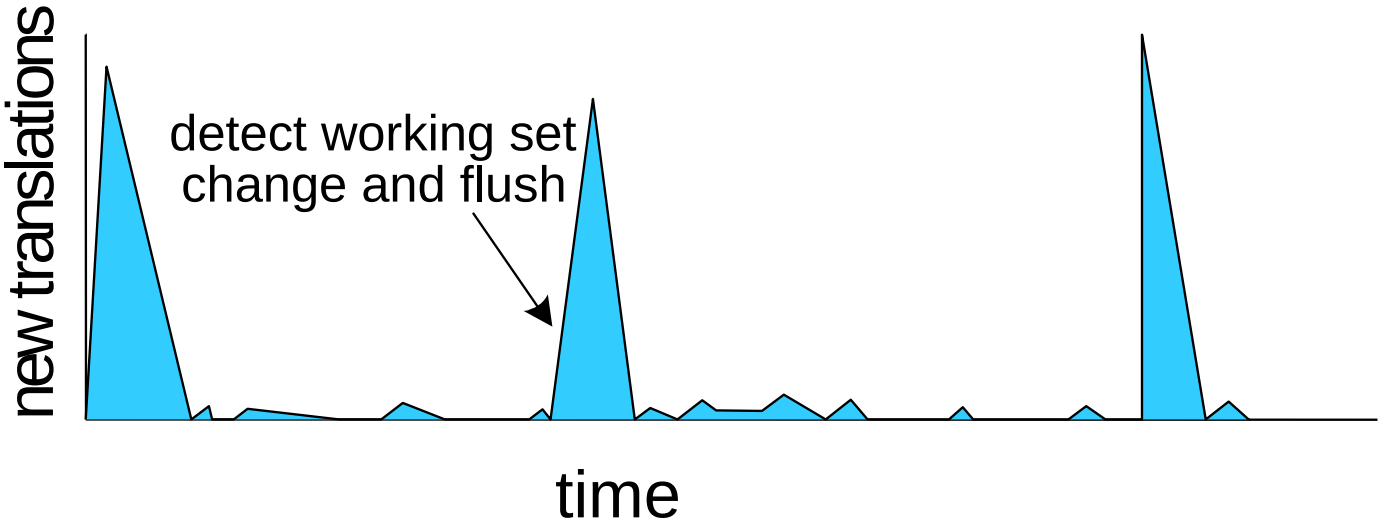
- Least recently used (LRU)
  - good is theory, problematic in practice
  - overhead of keeping track of the LRU block
  - *backpointers* are needed to eliminate chained links
  - fragmentation problem due to variable-sized blocks
  - unlink blocks before removing
    - maintain backpointers

Code Cache Back Pointers



Code Cache Replacement (2)

- Cache flush
  - when full or on phase change
  - gets rid of stale blocks
  - minimal maintainence overhead
  - even actively used blocks may be removed, and may need re-translation

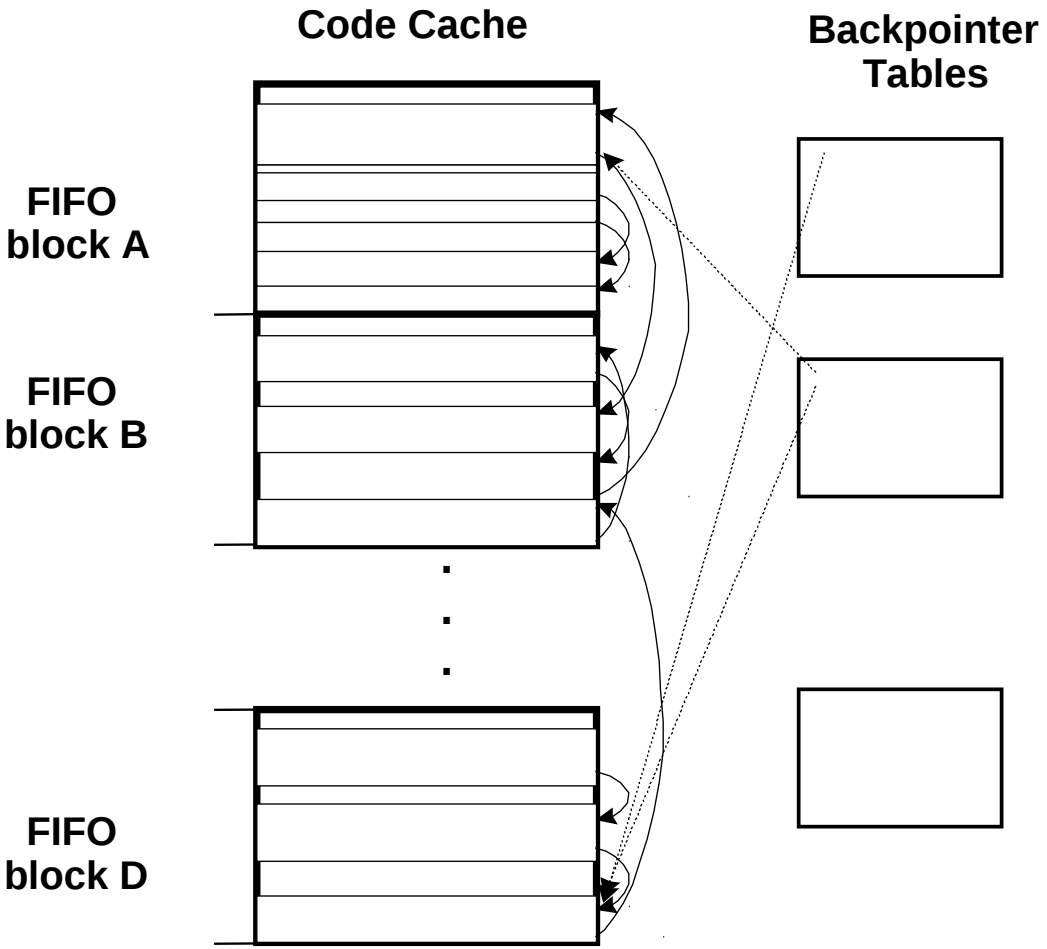


Code Cache Replacement (3)

- First In First Out (FIFO)
  - non-fragmenting, as cache can be maintained as a circular buffer
  - alleviates LRU problems at lower hit rates
  - needs to maintain backpointers
- Course-grained FIFO
  - partition code cache into large FIFO blocks
  - Links only maintained between blocks that span replacement boundaries (see Figure on next slide)

Code Cache Replacement (4)

- Course-grain FIFO (cont...)



PVM Performance

- Important for VM acceptance
  - optimization framework along with staged emulation
- Difference from static optimization
  - conservative, over small code regions, traces, superblocks
  - high level semantic information not available
  - profiling, architectural information can be used
- Will study in next chapter …