Process VM (2)

Outline

- Instruction Emulation
- Exception Emulation
- System Call Emulation
- Code Cache Management

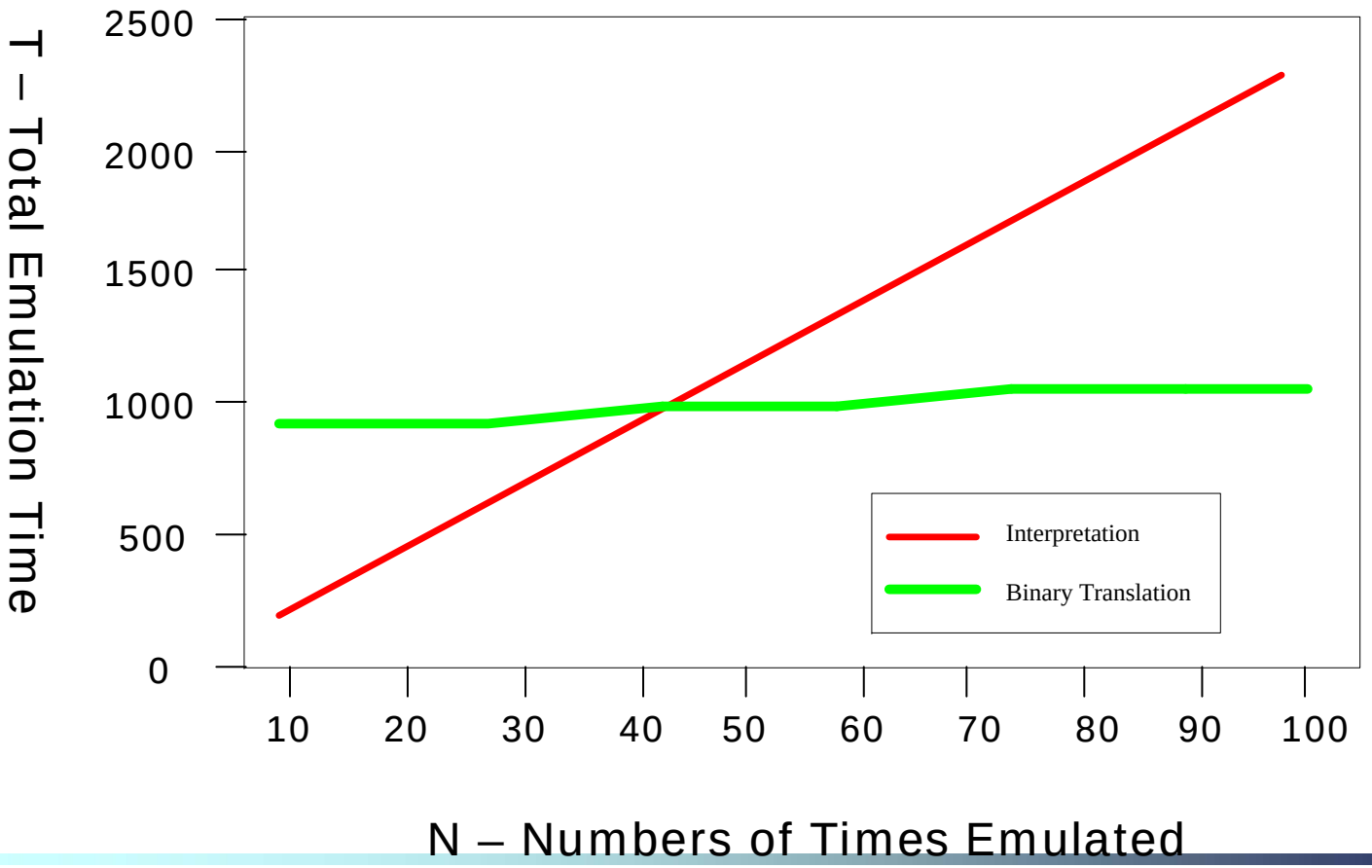
Instruction Emulation

- How to do high performance instruction emulation?
- Let us first understand the emulation overhead
- The overall time for emulating an instruction N times
  - T (Total) = S + NT
  - S: one-time start-up overhead
  - T: time required per emulation in steady state

	S	T
Interpretation	0	20
Binary translation	1000	2

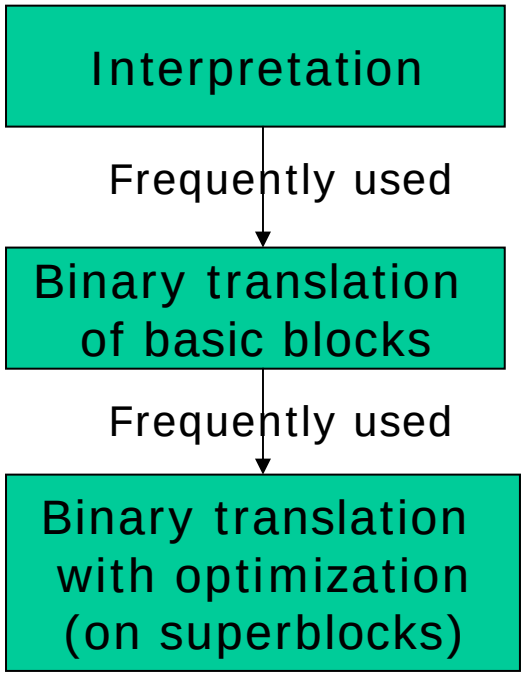
Performance Trade-off

- If N < 45, interpretation is better
- If N > 45, binary translation is better



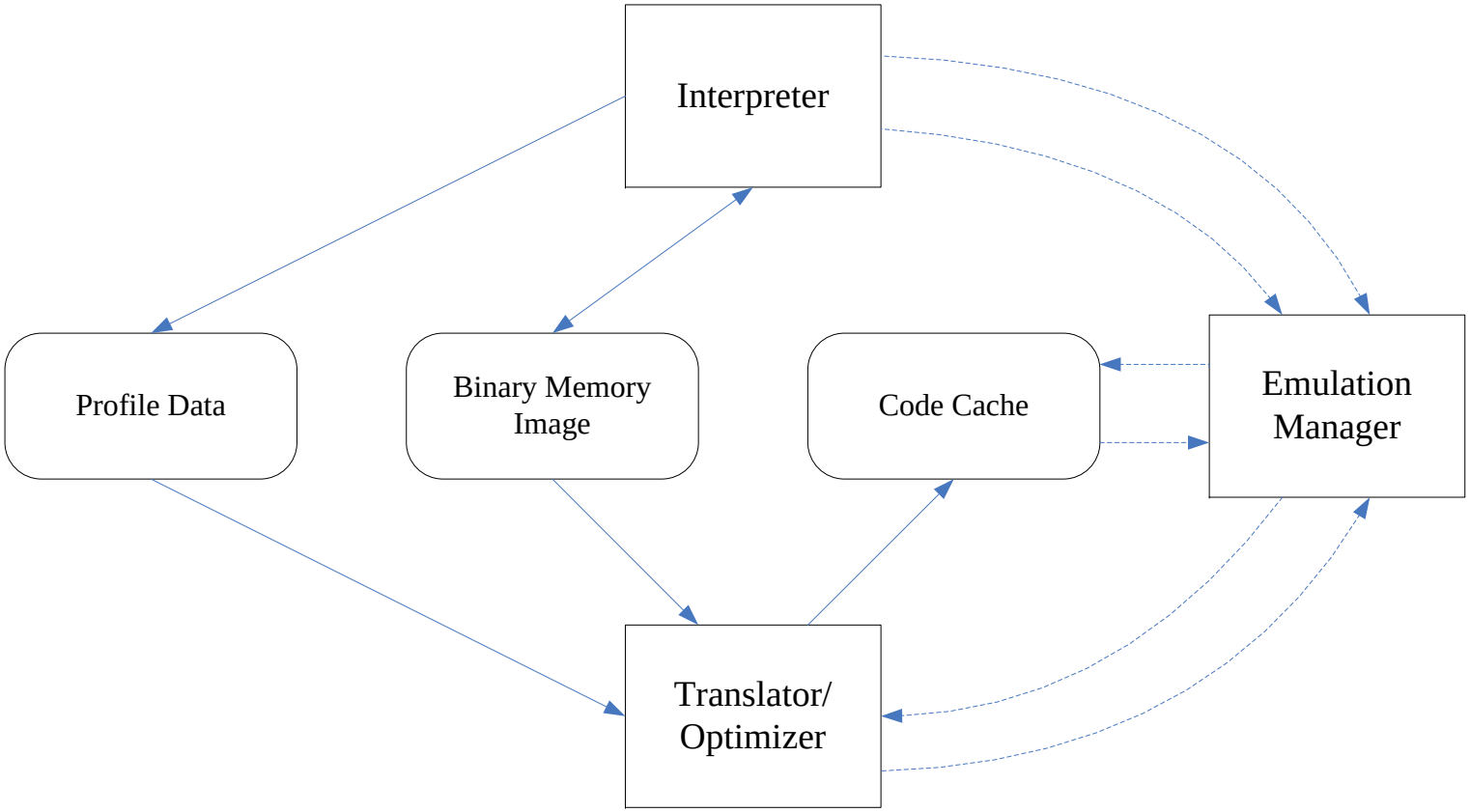
How to Exploit the Trade-Offs

- We cannot predict know how many times a block will be executed in advance
- But we know if it is executed frequently, (e.g., > 45), it is better to be translated
  - Why? It has a better chance of being a hot spot, so the start-up overhead can be offset
    - i.e., benefit of translation > cost of translation
- One solution: staged emulation
  - Begins emulation with interpretation
  - Collect *profile data* (e.g., execution frequency)
  - If it is a hot spot, translate it
  - If it is a real hot spot, translate it again with optimization with a larger block (e.g., superblock)

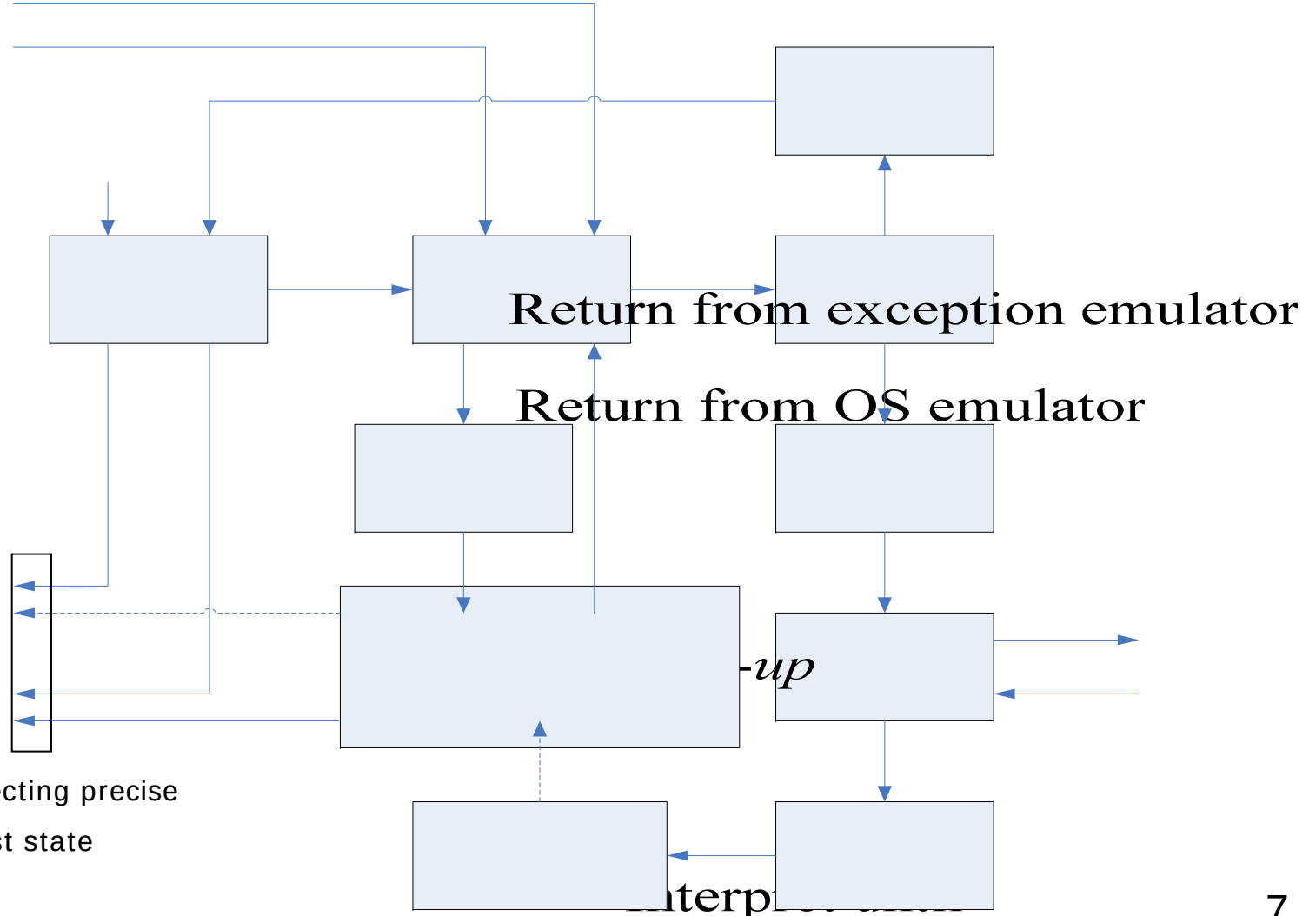


Instruction Emulation

- Frame work

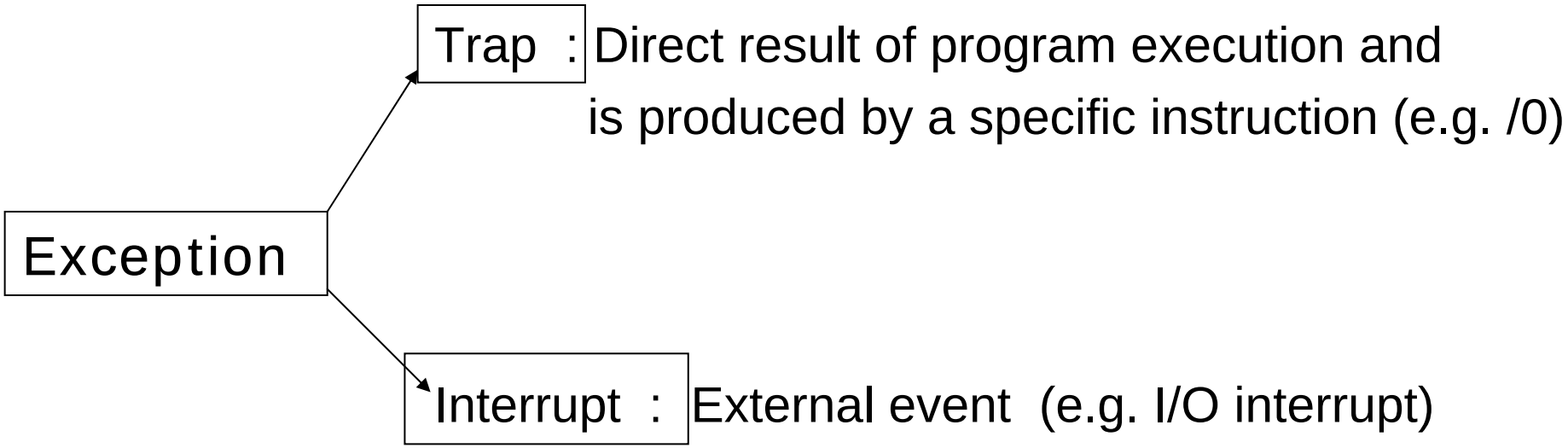


Staged Emulation Execution Flow



Exception Emulation

- Correct handling of exceptions raised during the emulation of guest process



Exception Emulation

- If there are exceptions raised during the emulation of guest process, we should emulate them correctly
- There are two types of exceptions
  - Trap: direct result of program execution and is produced b y a specific instruction
    - Divide by zero, memory access violation, page fault, etc.
  - Interrupt: external event
    - I/O interrupt, timer interrupt, etc.
- Actually, trap and interrupt are mechanism for transferring control from user program to OS
  - Another mechanism is system call which work similar to the trap except for argument passing

Taxonomy of Exceptions

- ABI visible:
  - All exceptions that are returned to the application via OS signa l
  - Function of OS and user-level ISA
  - E.g., if there is an OS signal for memory protection fault, then it is ABI visible
  - E.g., include exceptions that cause the application to terminate

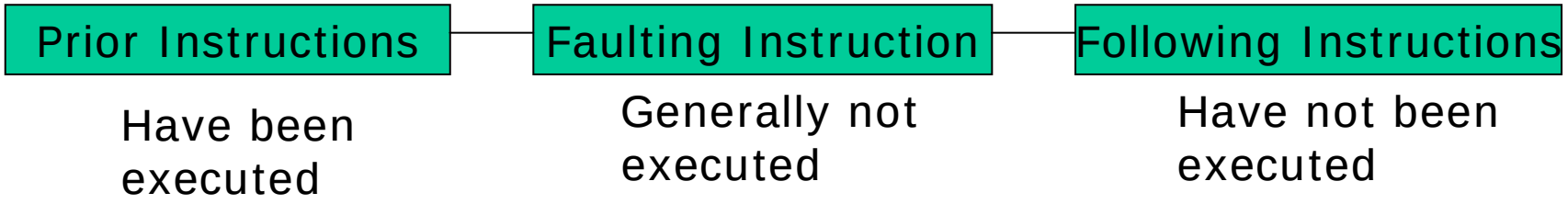
- ABI invisible:
  - ABI is essentially unaware of its existence
  - There are no signals for them or the application does not terminate when the exception occurs
  - E.g., timer interrupt or page fault

Some Exception Handling Basics

- When exception occurs, the following events occur
  - Execution ceases and the processor goes into a “precise” state
    - Called precise exception handling
  - Save PC and other registers
  - Go into privileged mode and OS gains control
  - OS saves remaining state of the faulting process
  - OS either handles the trap or jumps to the handling code
    - If the user established a trap handler, jump to the trap handler
  - After trap handling completes, the process’s precise state is restored and jump back to the faulting location of the process

Precise Exception Handling

- Most ISA supports precise exceptions
  - An exception is precise if the following conditions are true



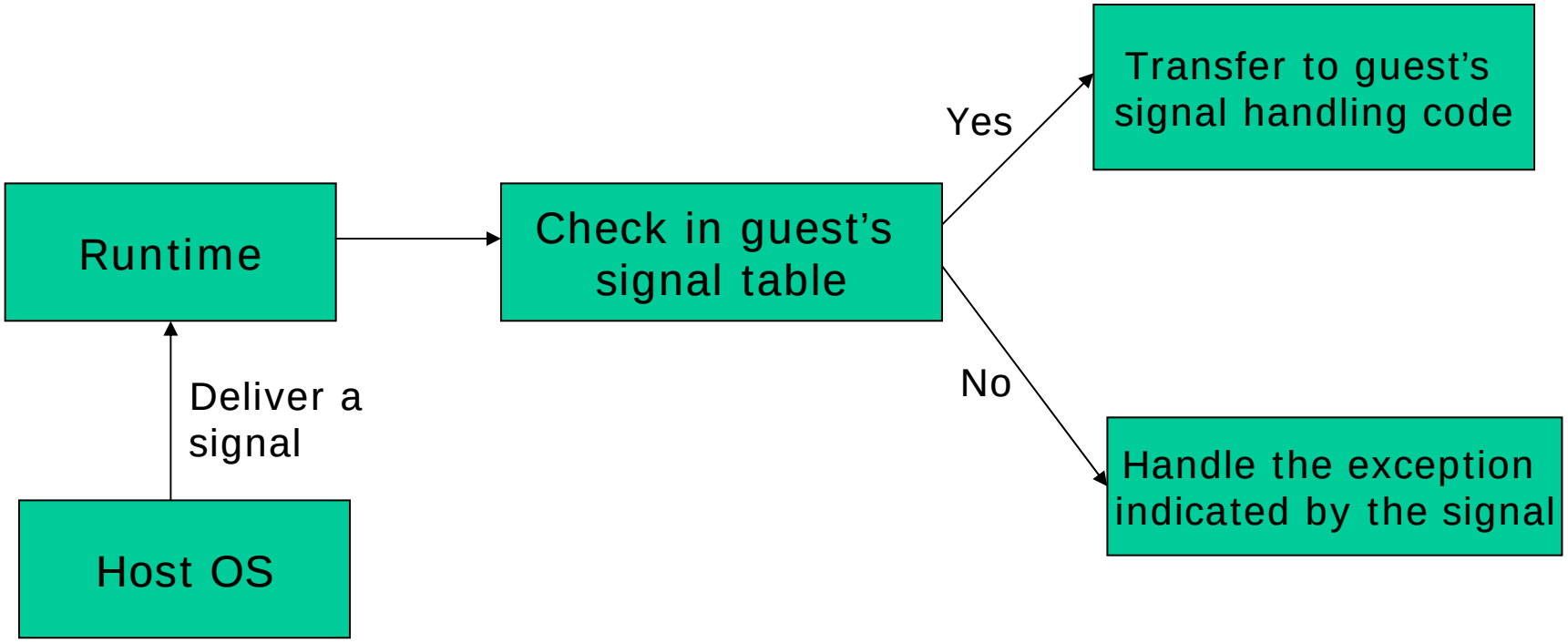
- Why precise exception?
  - Ensuring deterministic behavior when returned back

- Precise exception handling will be done automatically for the host process, but it should be done by the VM for the guest process

Exception Detection for a Trap

- Can be detected in two ways
  - Checked as part of an instruction's interpretation routine
    - E.g. check operands and the final sum a = b + c
    - Inefficient but can always be done
- Can be detected by host hardware during execution
  - As a result of executing emulation code (interpreted or translated)
  - OS signal mechanism play an important role
    - Runtime register all exceptions for which the host OS signals
    - When traps to the OS during emulation, the host OS will deliver signal
    - Runtime signal handler checks the signal and do appropriate action
  - What if the guest process itself registers signals via system call?
    - Runtime intercepts the call and register it as a "guest-registered" one
    - Runtime transfers to guest's signal handler when signal arrives

Signal Handling by Runtime



Three kinds of trapping condition

- Trap is visible both to guest ABI and host ABI
  - Invoke runtime signal handler
- Not visible to host ABI
  - Interpretive trap detection must be used
- Not visible to guest ABI
  - Extra traps: runtime trap handler will determine whether the trap condition would be visible to source instruction

Securing Precise Guest State

- The first thing is finding the excepting source PC
- Straightforward for interpretation
- When signal is delivered with a target PC, the current source PC points to the excepting source instruction

source state is clear when the exception signal comes

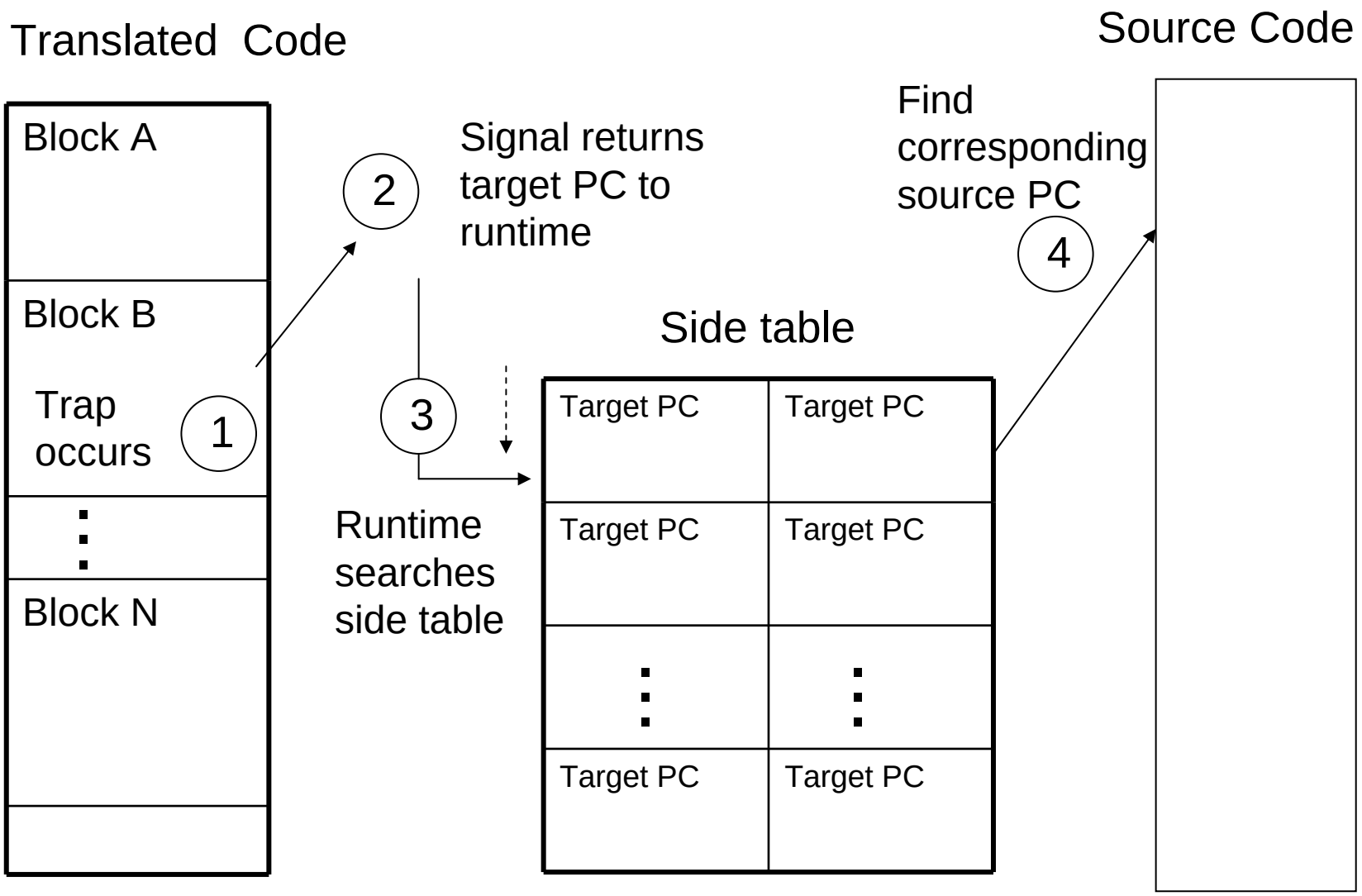
```
Add:
    ⋮
    sum =source1 + source2;
    regs[RT] = sum;
    PC = PC  + 4;
     ⋮
```

Securing Precise Guest State

- Not straightforward with binary translation
- Problem: binary translation typically does not keep a continuously updated version of source PC

- Use a reverse translation side table
  - Contains <target PC, source PC> pairs
  - Given the TPC of trapping instruction, runtime scan the table for a match and get the SPC

Finding the trapping source PC

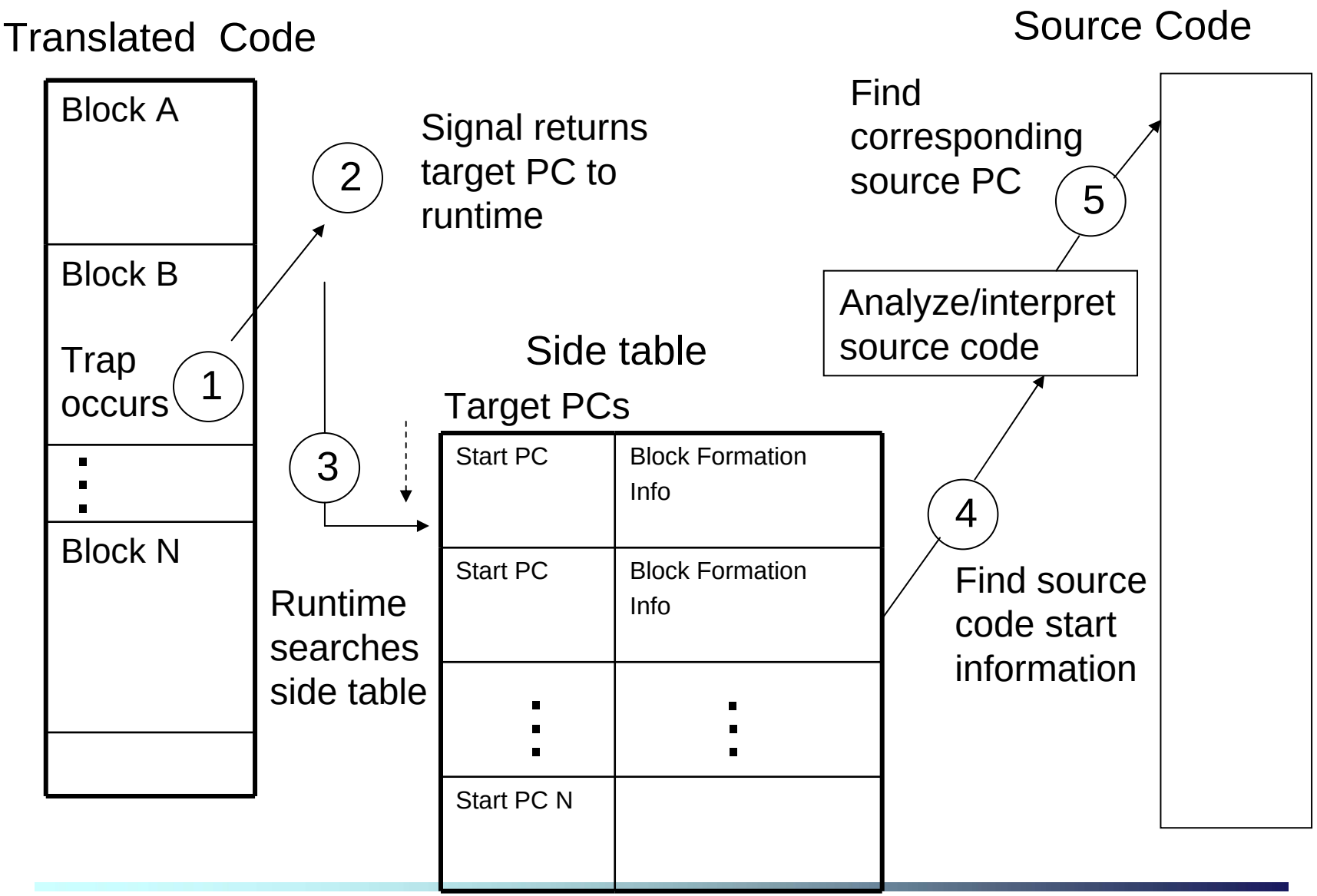


Inefficiency of Side Table

- The target PC lookup is expensive if we search linearly
- The table will be fairly large

- Solutions
  - Replace linear scan with binary search
  - Reduce table size by saving only the TPC and SPC for the beginning of the block, while other TPC and SPC are expressed as delta from these saved TPC and SPC
    - If the target ISA is a fixed size, save only SPC and its deltas
- Some complications (will be dealt with in Chap 4)
  - When a target instruction corresponds to multiple source instructions (e.g., load and add in RISC for a CISC instruction)
  - When the translated code is optimized and rearranged
  - Identify beginning of translated source block and analyze/interpret

Using an optimized side table



Precise State of Registers & Memory

- When an exception occurs during execution of translated block, the precise state of registers and memory should also be restored
- As to registers, no code reordering, no removal of register updates, and consistent register mapping guarantees recovering source register state from target register state
- As to memory, it is changed only by store instructions and if the translated code keeps the order of source code store instructions, the memory state is consistent if trapped
- If there are code reordering, then we need to handle them

Operating System Emulation

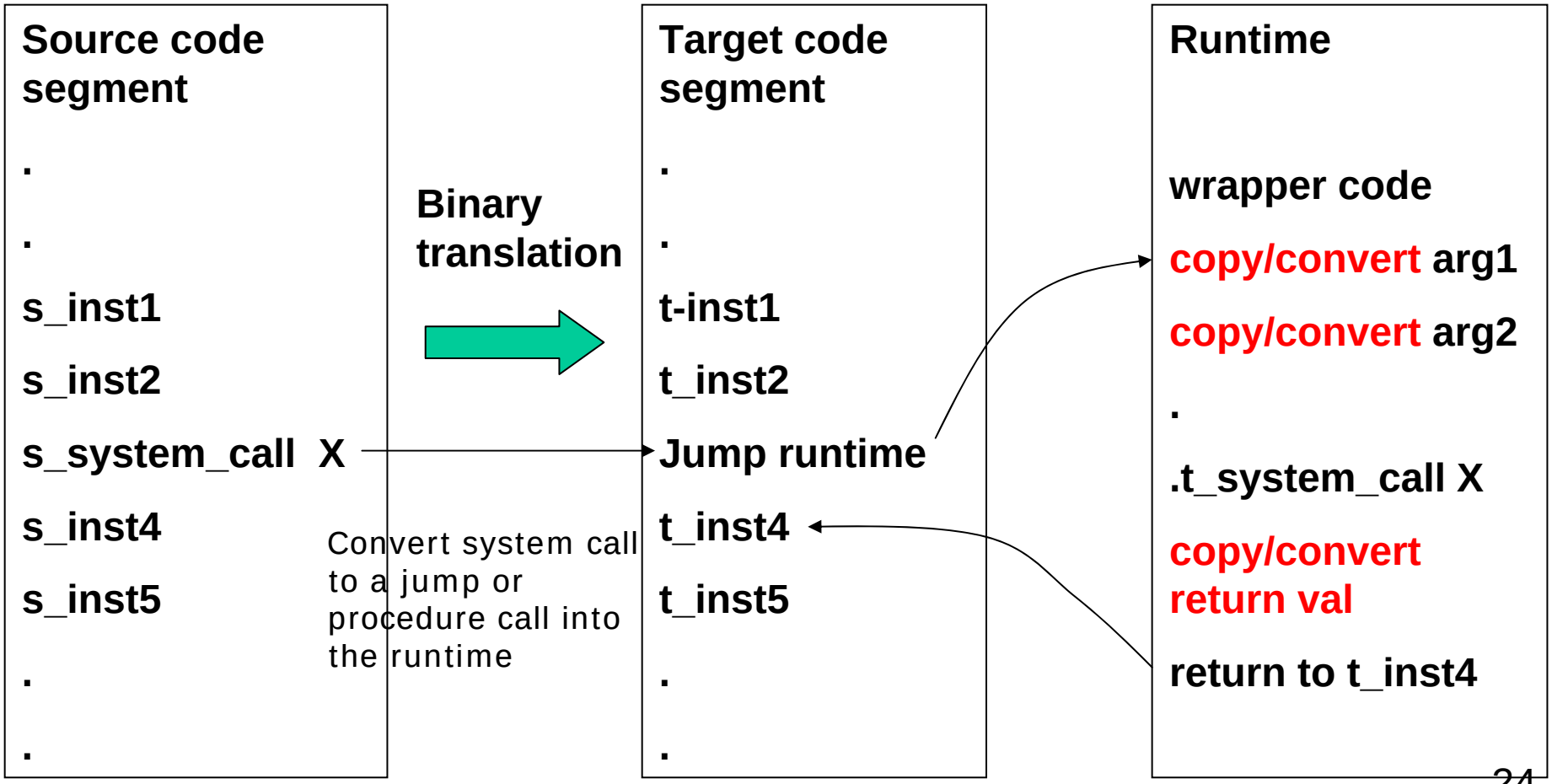
- Process-level VM (ABI level compatible) does not emulate the individual instructions in the guest's OS code
- Just emulate the function of the guest's system calls
  - By converting them to host OS calls
    - Mostly formatting arguments and return values
    - E.g., system calls in IA-32 pass arguments via memory while those in RISC pass arguments via registers
- Two cases:
  - Guest and Host OS are the same
  - Guest and Host OS are the different

OS Call Translation for the Same OS

- It may be necessary to
  - move and format arguments and return values
  - form some data conversions

- Reason
  - Because of the hardware differences of the target platforms
  - Arguments passing will be done in a memory resident stack or registers
  - Above changes must be compensated when emulating a sys tem call

OS Call Translation for the Same OS



Runtime-implemented OS functions

- Some system calls may be handled directly be the runtime

- Examples
  - System call to establish a signal on behalf of the guest application
    - Runtime record the application's signal in a side table and return to the guest process

  - System call for memory management
    - Process VM always maintain control on overall memory management
    - E.g. Linux brk() system call is handled directly by runtime

Code Cache Management

- Code cache differs from a H/W cache in three ways
  - The cached blocks do not have a fixed size
    - Depends on the size of the translated target block
  - The cache blocks are dependent on another due to chaining
    - If a block is removed, corresponding link pointers must be updated
  - There is no copy of the cache contents in a backup space
    - Need to be regenerated if want it back
- Need different cache management algorithms

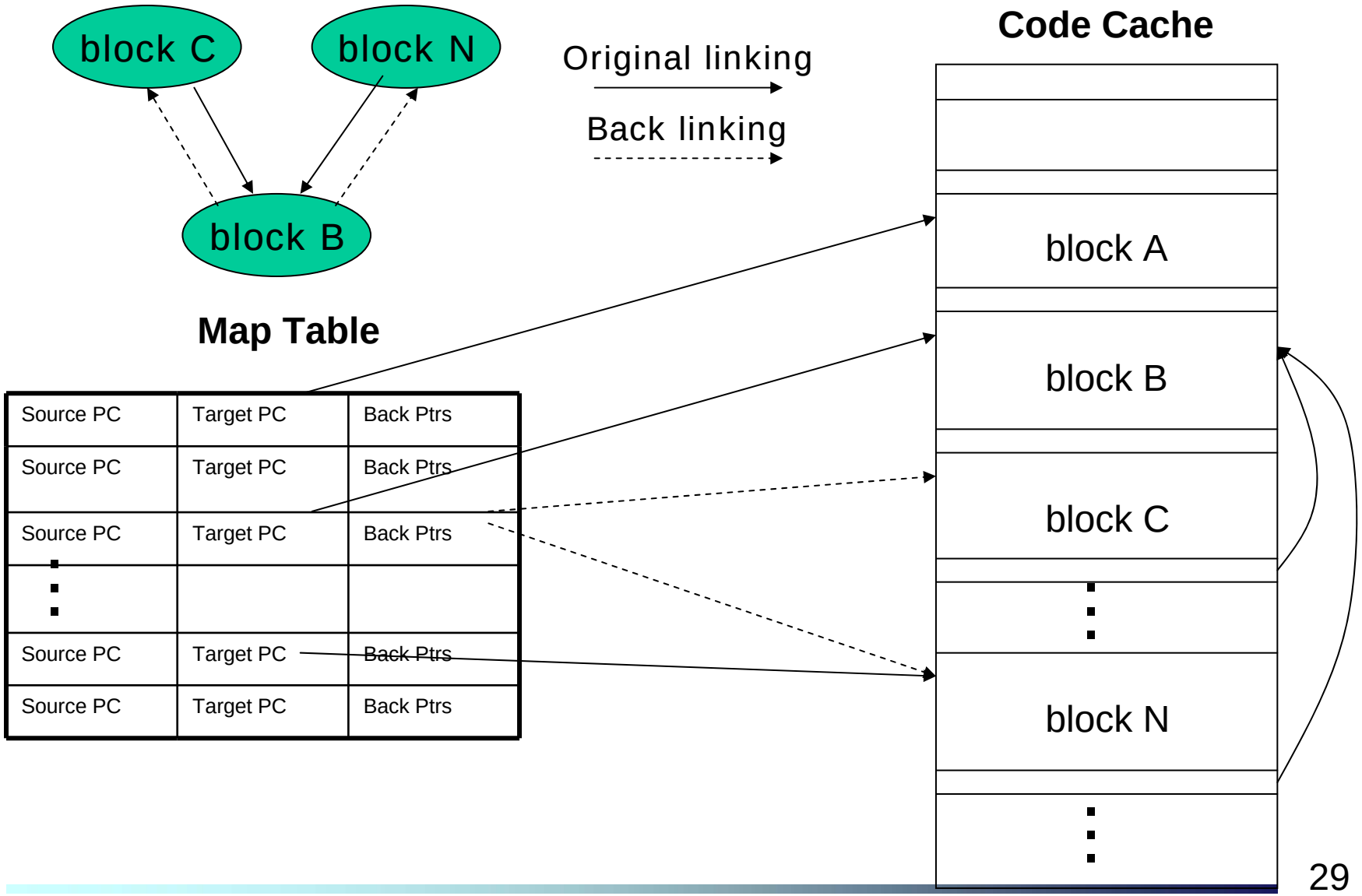
Code Cache Implementations

- Two key operations involving the code cache
  - Given a SPC, find TPC
    - When control transfer to code cache
    - Use SPC-to-TPC map table
  - Given a TPC, find a SPC
    - Used to find the precise source PC when exception occurs
    - Using reverse translation side table

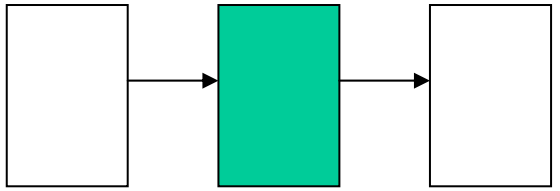
Replacement algorithms

- Least recently used (LRU)
  - Replaces a block that was unused for the longest period of time
  - Make use of temporal locality
- Problems in implementation
  - Overhead in keeping track of the least recently used block
    - Need to update LRU information each time a block is entered
  - When an arbitrary block is removed, any link pointers linked to it must be updated
    - Backpointers can help for updating link pointers
  - Cache fragmentation issue when an LRU block is removed
- LRU is not used for code cache replacement

The use of backpointers



Flush-When-Full Algorithm

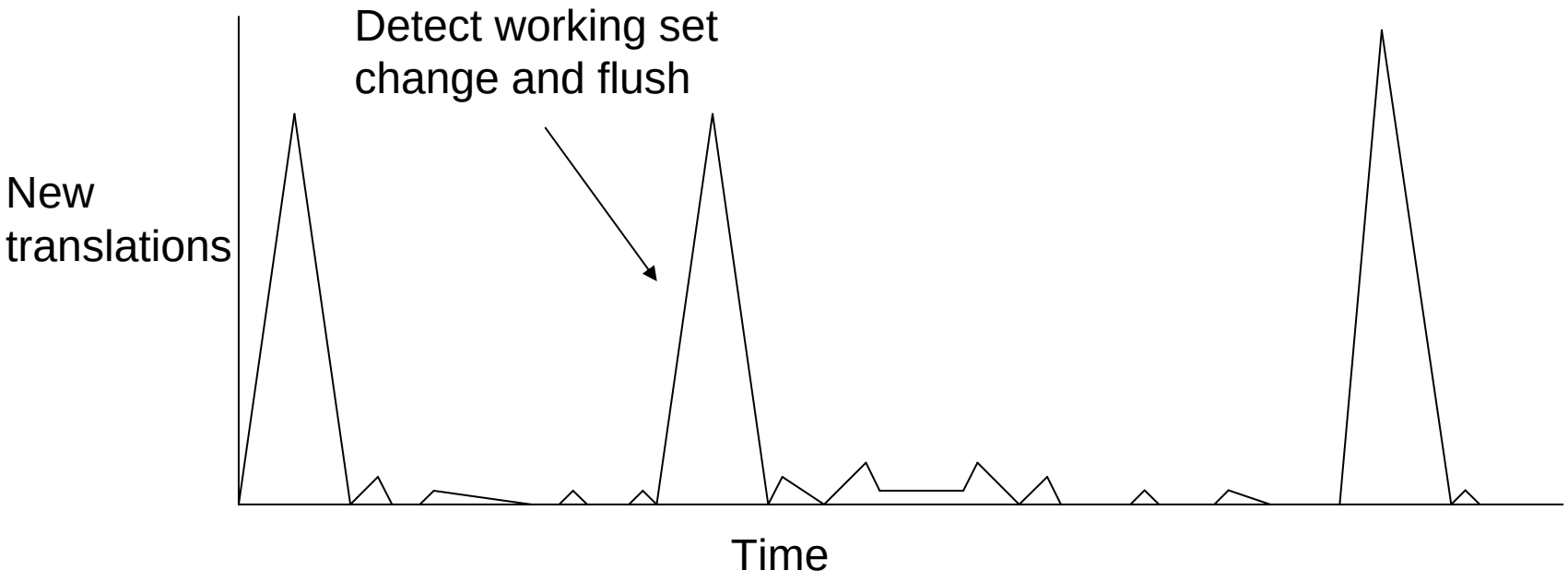


- Start from scratch approach
- Advantages
  - Provides an opportunity to eliminate superblocks whose control paths are changed over time
- Disadvantages
  - All blocks being actively used have to be retranslated after flush

Preemptive Flush

- Based on an observation
  - Programs operate in phases
  - A phase change is usually associated with an instruction working set change
- Preemptive flush
  - When an increase in new translation is detected, the entire code cache is flushed to make room for a new working set

Preemptive Flush



Fine-Grained FIFO

- A nonfragmenting algorithm that exploits temporal locality
  - Code cache is managed as a circular buffer
    - Oldest blocks are removed to make a room for new ones

  - Reverse-translation table can also be managed in a FIFO manner

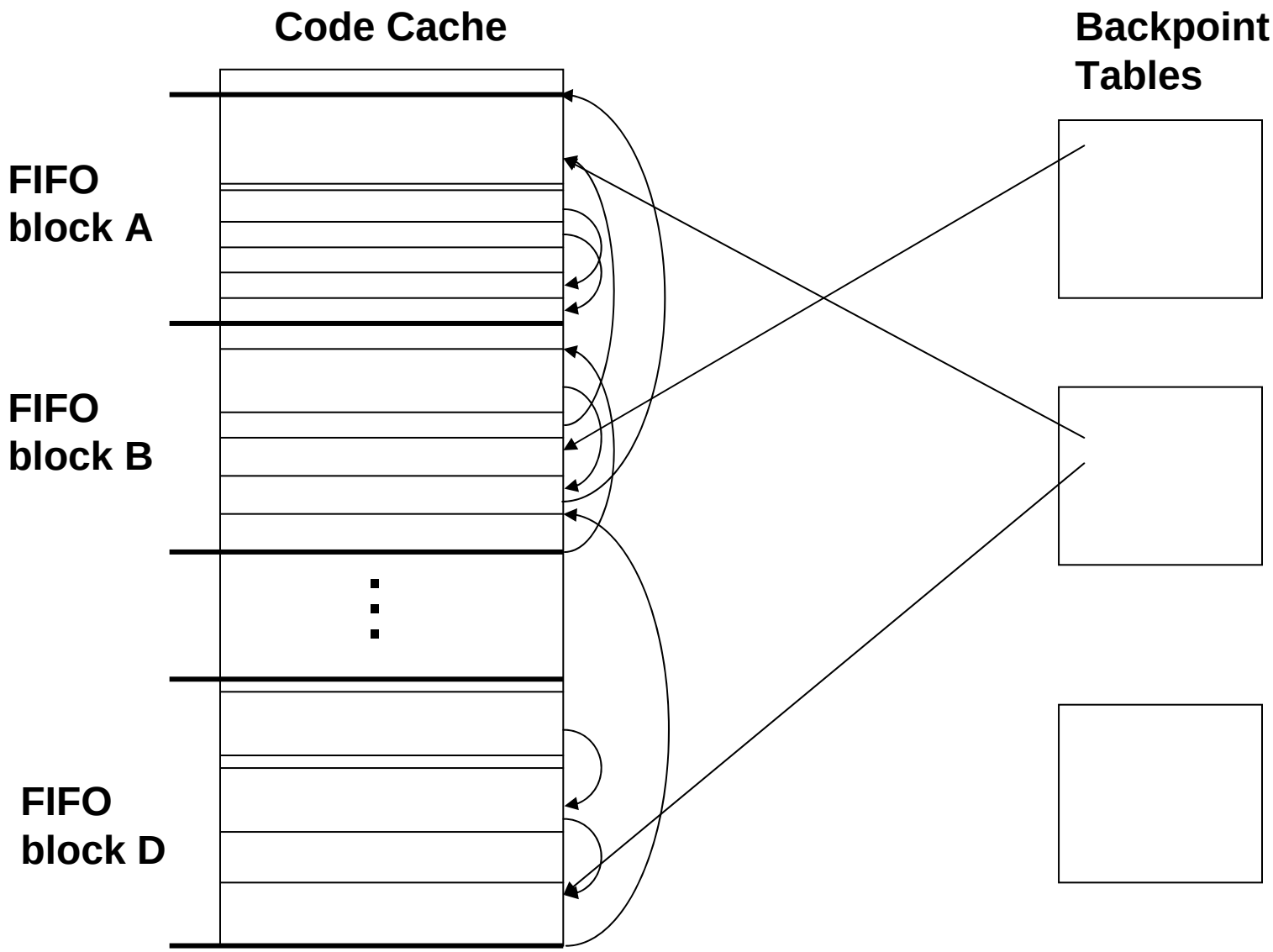
- Performance
  - Overcomes some disadvantages of LRU
  - Still need backpointers to keep track of chaining

Coarse-Grained FIFO

- This is for simplifying backpointers
- Partition the code cache into very large FIFO blocks
  - Each FIFO block is allocated sequentially as usual, but when the code cache is full, the oldest FIFO block is removed as a whole

- Why is this good for simplfying backpointers?
  - Maintain the backpointers only on a FIFO block basis
    - Do not keep the backpointers within the same FIFO block
    - Only keep them for different FIFO blocks
      - ✓ Why? Translation blocks in the same FIFO block will be removed together
  - When a FIFO block is removed, all its backpointers are handled
  - The point is that the number of intra-FIFO block links is much higher than inter-FIFO block links

Coarse-Grained FIFO Picture



Performance comparison