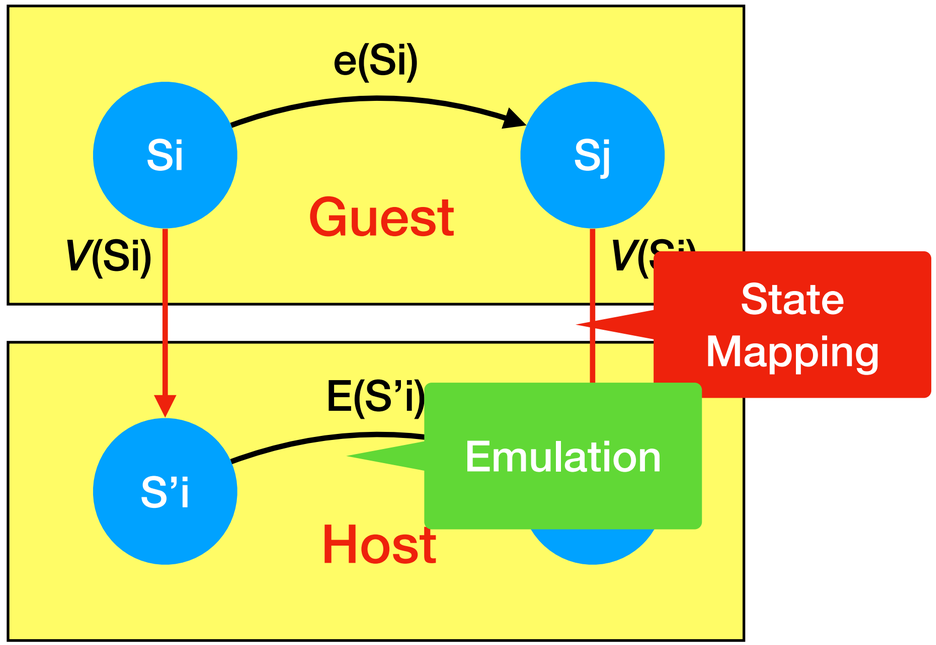
Process VM: Interpretation and Binary Translation

Prof. Shih-Wei Li

Department of Computer Science and Information Engineering
National Taiwan University

Review: Virtualizing ISA

- Guest ISA != host ISA:
  - The guest and host instructions/registers do not match
  - Guest programs cannot execute directly on the host hardware
  - Translate guest instructions (binary translation) or interpretation



Emulation

- Definition: the process of implementing the interface and functionality of one system or subsystem on a system or subsystem having a different interface and functionality
- Instruction set emulation is a key aspect of many VM implementations
  - VM must support a program binary compiled for an instruction set that is different from the one implemented by the host processor(s)
  - Avoid if possible for performance overhead — system VMs execute instructions natively with hardware virtualization support

Emulation

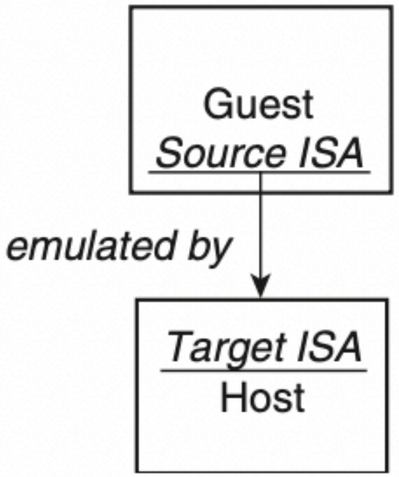


Figure 2.1 Terms Describing the Emulation Process. *Emulation allows a guest to support a source instruction set while running on a host platform executing a target instruction set.*

Emulation

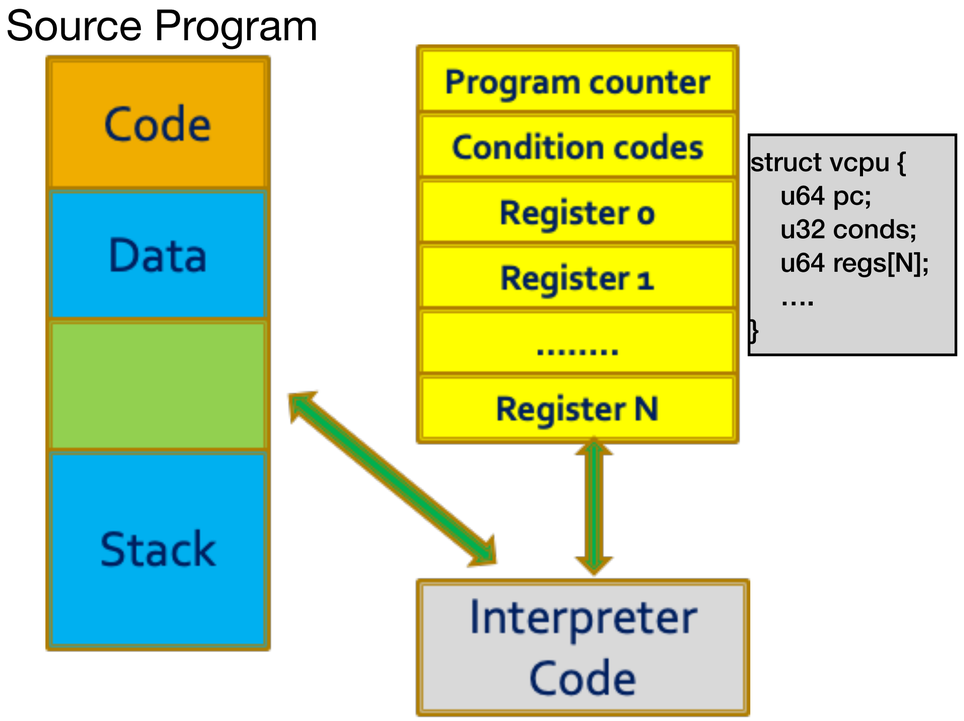
- Terminology:
  - Source ISA/program: the ISA/program to be emulated
  - Target ISA/program: the underlying ISA/program that is actually executed
- Two ways to carry out instruction set emulation:
  - ***Interpretation***: translate single instruction at a time
  - ***Binary Translation***: translate a block of instructions at a time; better solution for repeated instruction executions

Agenda

- Emulation
- **Interpretation**
- Binary Translation
- Case Study of Binary Translation — QEMU

Interpretation

- Source ISA states (CPU registers and memory) are held in the interpreter's memory



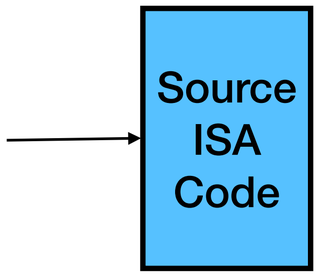
Interpretation

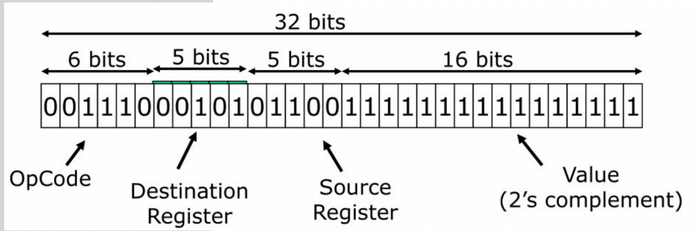
- Decode-and-dispatch interpreter:
  - Step through the source program, instruction by instruction, reading and modifying the source state according to the instruction
  - Simple implementation: to interpret instructions in a loop
  - Includes two phases: decode and dispatch

Interpretation

Pseudo code for “decode-and-dispatch” interpretation of PowerPC source

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





From: http://www.cse.unsw.edu.au/~cs2121

```
LoadWordandZero(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;
}
```

Interpretation

- High emulation cost with with decode-dispatch:
  - Requires executing a lot more target instructions to execute: a single *Load Word and Zero* instruction could involve the execution of tens of instructions in the target ISA
  - Potentially involve jumps and branches, loads/stores, and shifts/masks

Interpretation: Improving the Dispatch Loop

- Threaded Interpretation: Append a portion of the dispatch code [to look a dispatch table] to the end of each instruction interpretation routine; why is this helpful?

```
LoadWordAndZero(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;
}
```

```
if (halt || interrupt) goto exit;
inst=code[PC];
opcode=extract(inst,31,6)
routine=dispatch[opcode];
```

Interpretation: Improving the Dispatch Loop

- Threaded Interpretation v.s. decode-and-dispatch

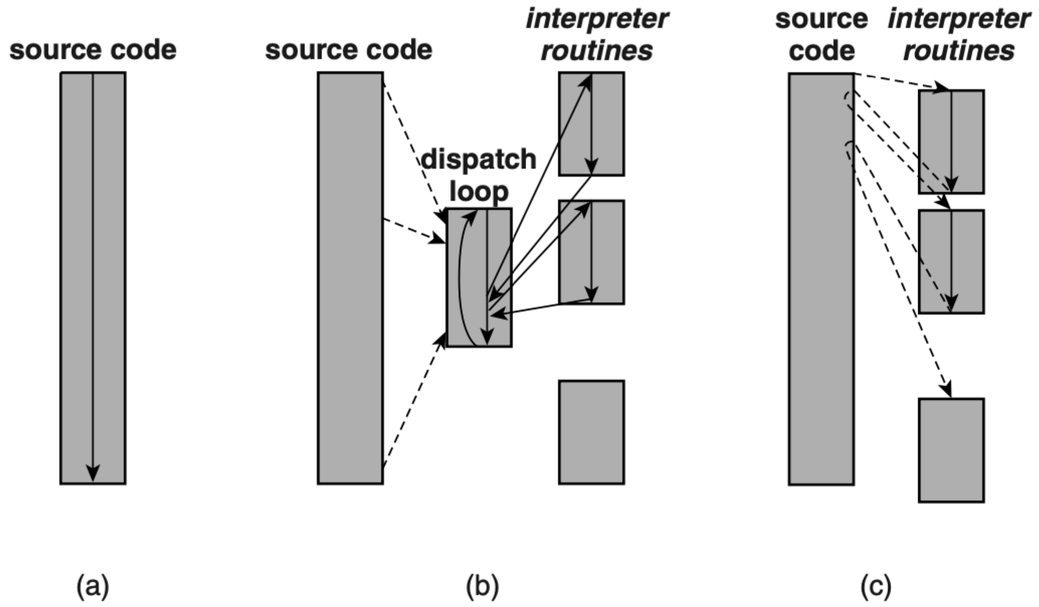


Figure 2.5 Interpretation Methods. *Control flow is indicated via solid arrows and data accesses with dotted arrows. The data accesses are used by the interpreter to read individual source instructions. (a) Native execution; (b) decode-and-dispatch interpretation; (c) threaded interpretation.*

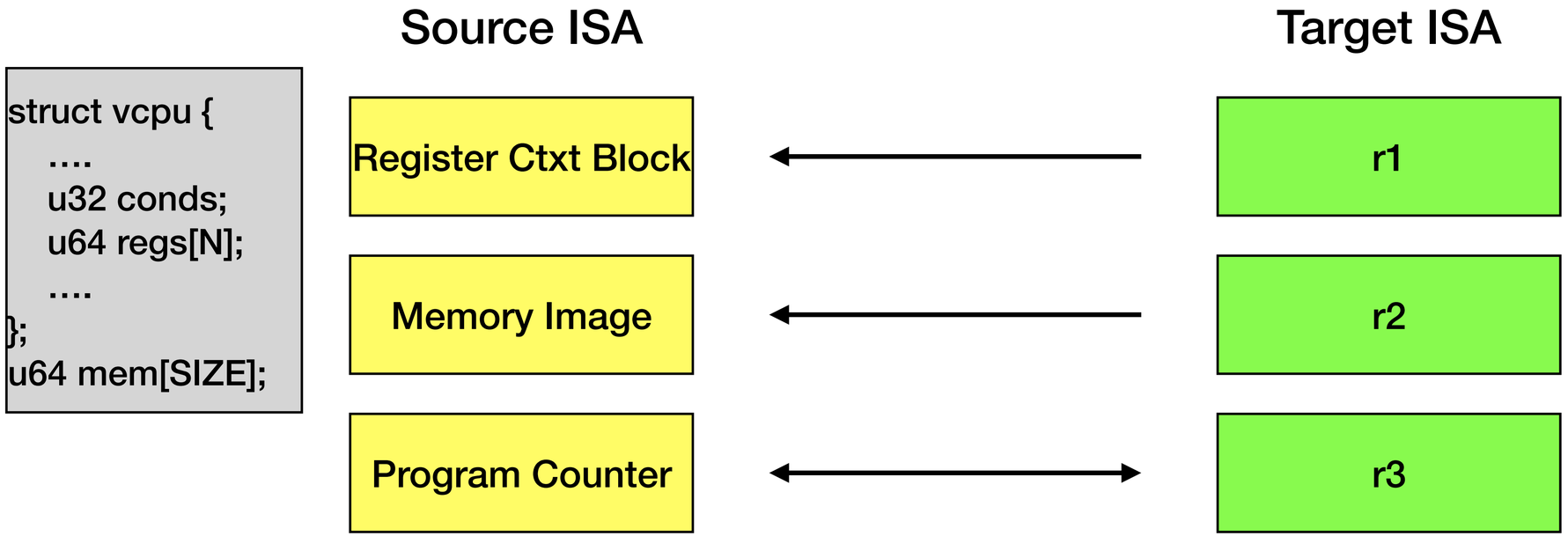
Agenda

- Emulation
- Interpretation
- **Binary Translation**
- Case Study of Binary Translation — QEMU

Binary Translation

- The process of converting the source binary program into the target binary program with customized code
- Generate custom code for every source instruction
  - Get rid of the overhead in repeated parsing, decoding, and jumping
  - Do not usually have to regenerate once generated already
- Integrate techniques to optimize the binary translation

Binary Translation: Register Mapping



Binary Translation: Register Mapping

From IA-32 code to PowerPC code

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

```
struct vcpu {
    ....
    u32 conds;
    u64 regs[N];
    ....
};
u64 mem[SIZE];
```

```
r1 points to IA-32 register context block
r2 points to IA-32 memory image
r3 contains IA-32 ISA PC value

lwz     r4,0(r1)      ;load %eax from register block
addi    r5,r4,4       ;add 4 to %eax
lwzx    r5,r2,r5      ;load operand from memory
lwz     r4,12(r1)     ;load %edx from register block
add     r5,r4,r5      ;perform add
stw     r5,12(r1)     ;put result into %edx
addi    r3,r3,3       ;update PC (3 bytes)

lwz     r4,0(r1)      ;load %eax from register block
addi    r5,r4,4       ;add 4 to %eax
lwz     r4,12(r1)     ;load %edx from register block
stwx    r4,r2,r5      ;store %edx value into memory
addi    r3,r3,3       ;update PC (3 bytes)

lwz     r4,0(r1)      ;load %eax from register block
addi    r4,r4,4       ;add immediate
stw     r4,0(r1)      ;place result back into %eax
addi    r3,r3,3       ;update PC (3 bytes)
```

Binary Translation: Register Mapping

From IA-32 code to PowerPC code

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

```
r1 points to IA-32 register context block
r2 points to IA-32 memory image
r3 contains IA-32 ISA PC value

lwz     r4,0(r1)      ;load %eax from register block
addi    r5,r4,4       ;add 4 to %eax
lwzx    r5,r2,r5      ;load operand from memory
lwz     r4,12(r1)     ;load %edx from register block
add     r5,r4,r5      ;perform add
stw     r5,12(r1)     ;put result into %edx
addi    r3,r3,3       ;update PC (3 bytes)

lwz     r4,0(r1)      ;load %eax from register block
addi    r5,r4,4       ;add 4 to %eax
lwz     r4,12(r1)     ;load %edx from register block
stwx    r4,r2,r5      ;store %edx value into memory
addi    r3,r3,3       ;update PC (3 bytes)

lwz     r4,0(r1)      ;load %eax from register block
addi    r4,r4,4       ;add immediate
stw     r4,0(r1)      ;place result back into %eax
addi    r3,r3,3       ;update PC (3 bytes)
```

```
struct vcpu {
    ....
    u32 conds;
    u64 regs[N];
    ....
};
u64 mem[SIZE];
```

Lots of memory accesses!

Binary Translation: Register Mapping

- Lots of memory accesses involved in the previous showcase — cause a slowdown!
  - How can we do better? Adopt a different register mapping scheme
- Easier if the number of target registers > the number of source registers
  - Map one source register to a respective target register
  - Example: translating an x86 binary for a RISC machine
- The mapping may be performed on a per-block, per-trace, or per-loop basis
- Infrequently used registers may not need to be mapped

Binary Translation: Register Mapping

Binary Translation: Register Mapping

From IA-32 code to PowerPC code

```
r1 points to IA-32 register context block
r2 points to IA-32 memory image
r3 contains IA-32 ISA PC value
r4 holds IA-32 register %eax
r7 holds IA-32 register %edx
```

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

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

Binary Translation: Static Translation

- Static Translation: translate a program in its entirety before beginning emulation
  - Less overhead during actual execution
- Problem:
  - Code discovery and code location are issues!

Binary Translation: Code Discovery Problem

- Code discovery is difficult — how to identify the starting point of all instruction sequences before running the code?
  - Compilers may intersperse data with code
  - Instructions are of variable-length in CISC ISAs like x86

Binary Translation: Code Discovery Problem

- Easier for fixed instruction ISA
  - All instruction boundaries are clearly identified
  - Unfortunately, the most popular ISA (x86 and ARM/Thumb) are all variable length

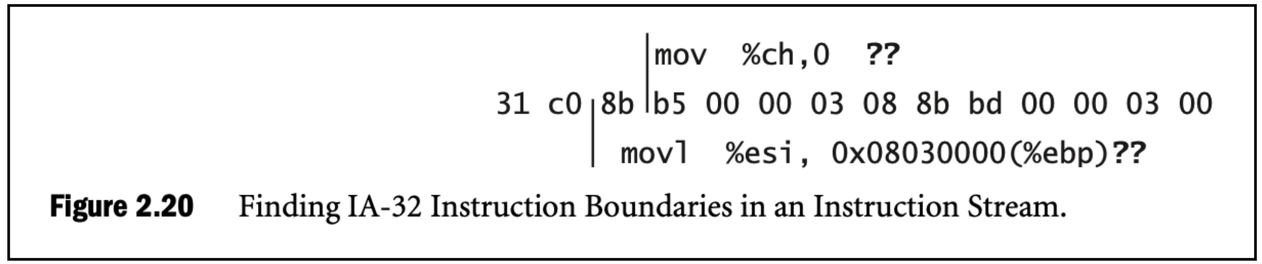


Figure 2.20 Finding IA-32 Instruction Boundaries in an Instruction Stream.

Binary Translation: Code Location Problem

- Examples of jumps and indirect jumps:

```
jne   .L6
jle  .L6
jmp  .L6
```

Direct Jumps

```
jmp   %eax
jmp  .L6(,%eax,4)
ret
```

Indirect Jumps

Binary Translation: Code Location Problem

- TPC (target PC) is different from SPC (source PC)
- For direct jumps/branches (jump to a fixed location), the binary translator can tell which TPC the SPC is mapped to (can do so in static time)
- For indirect jumps/branches, SPC is unknown to the binary translator in static time
  - Imagine the target (ex: x2) of a branch instruction (ex: “bl x2” on Arm) can only be resolved at run time: where does x2 point to? is the code pointed by x2 valid?

Binary Translation: Code Location Problem

```
mov1    %eax, 4(%esp)       ;load jump address from memory
jmp     %eax                ;jump indirect through %eax
```

(a)

Unknown in static time

```
addi    r16,r11,4           ;compute IA-32 address
lwzx    r4,r2,r16           ;get IA-32 jump address from IA-32 memory image
mtctr   r4                  ;move to count register (ctr)
bctr                        ;jump indirect through ctr
```

(b)

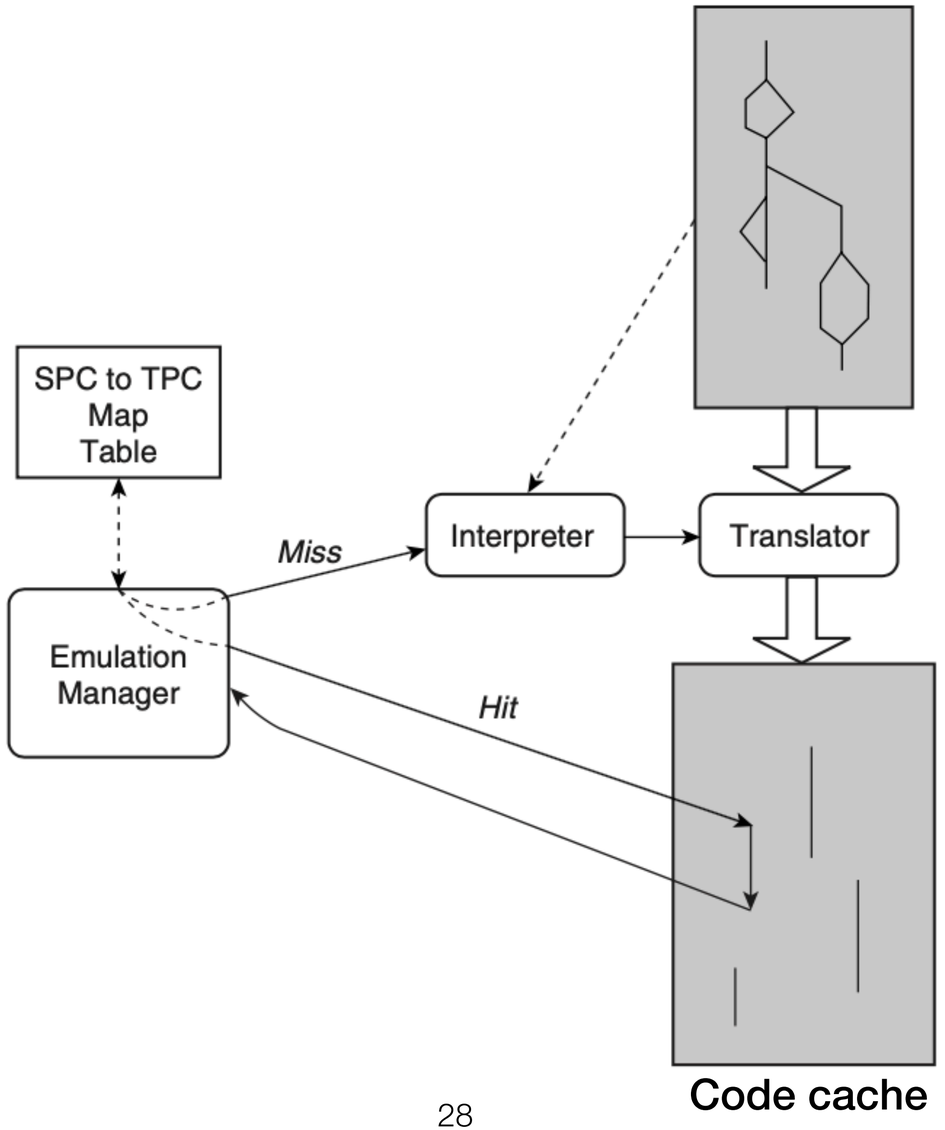
Figure 2.22 Translation of an Indirect Jump Code Sequence. *The value in the jump register is a source code address in both cases; consequently, this code translation will not work correctly. (a) IA-32 source code; (b) PowerPC target code.*

Binary Translation: Dynamic Binary Translation (DBT)

- General solution for code discovery is to ***dynamically*** translate the binary while the program is operating on actual input data

- New code is generated ***incrementally*** as the execution proceeds

Binary Translation: Dynamic Binary Translation (DBT)



Binary Translation: Dynamic Binary Translation (DBT)

- Emulation Manager (EM) provides high-level control for DBT at run time
- The interpreter and the binary translator generate the target code and put it in a host memory region called the ***code cache***
- DBT builds an address ***map table*** and updates on the fly
  - Associates the SPC for a block of source code with the TPC for the respective block of the translated code
  - The size of the table is dependent on the dynamic code regions executed
  - A map table is typically implemented as a hash table
  - The EM queries the map table to index the code cache

Binary Translation: Dynamic Binary Translation (DBT)

- The system translates one block of source code called *dynamic basic block* at a time
  - A dynamic basic block is determined by the actual flow of a program as it is executed
  - A dynamic basic block always ***begins*** at the instruction executed immediately after a branch or jump, follows the sequential instruction stream, and ***ends*** with the next branch or jump

Binary Translation: Dynamic Binary Translation (DBT)

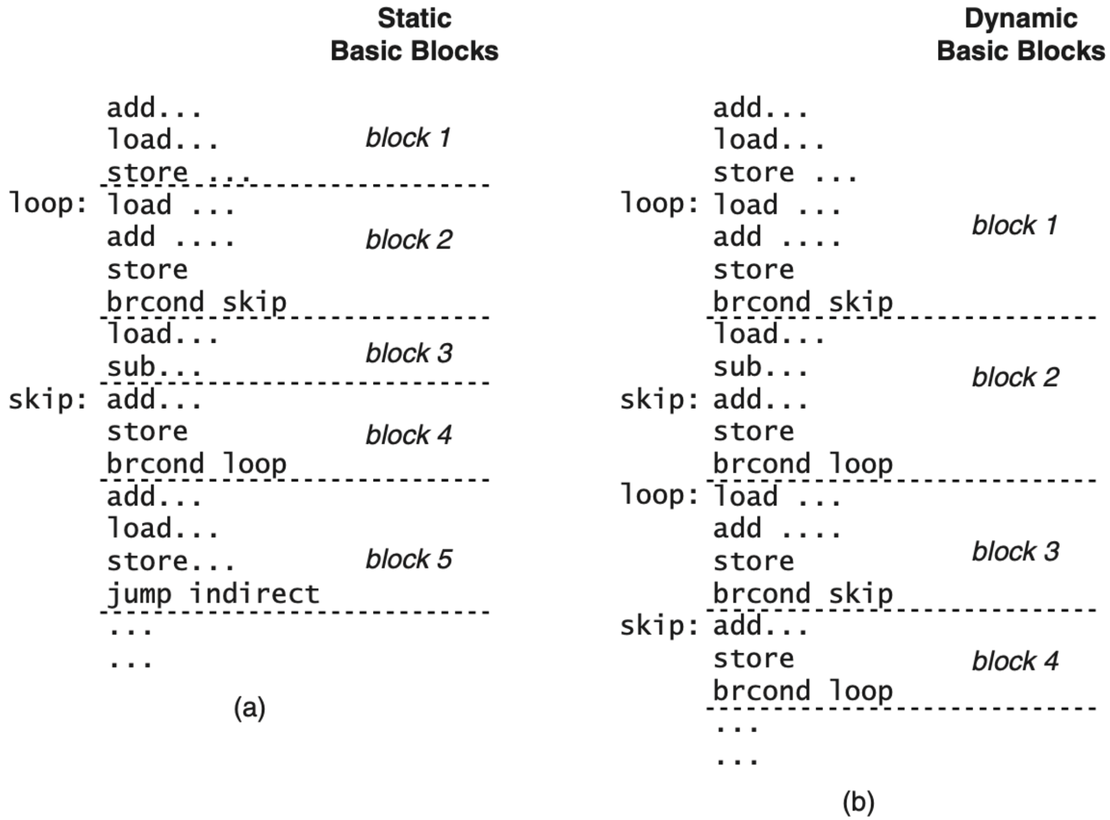
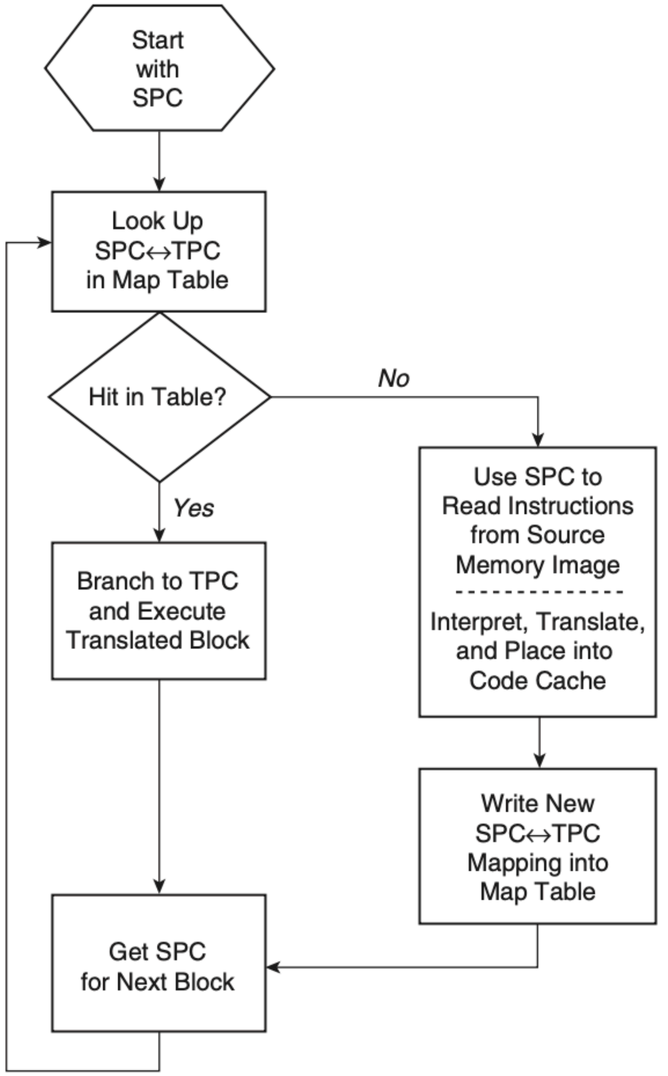


Figure 2.24 Static Versus Dynamic Basic Blocks. *(a) Static basic blocks are code sequences with one entry point and one exit point. They begin and end with control transfer instructions or targets of control transfer instructions. (b) Dynamic basic blocks are often larger than static basic blocks and are determined by the actual flow of control at run time.*

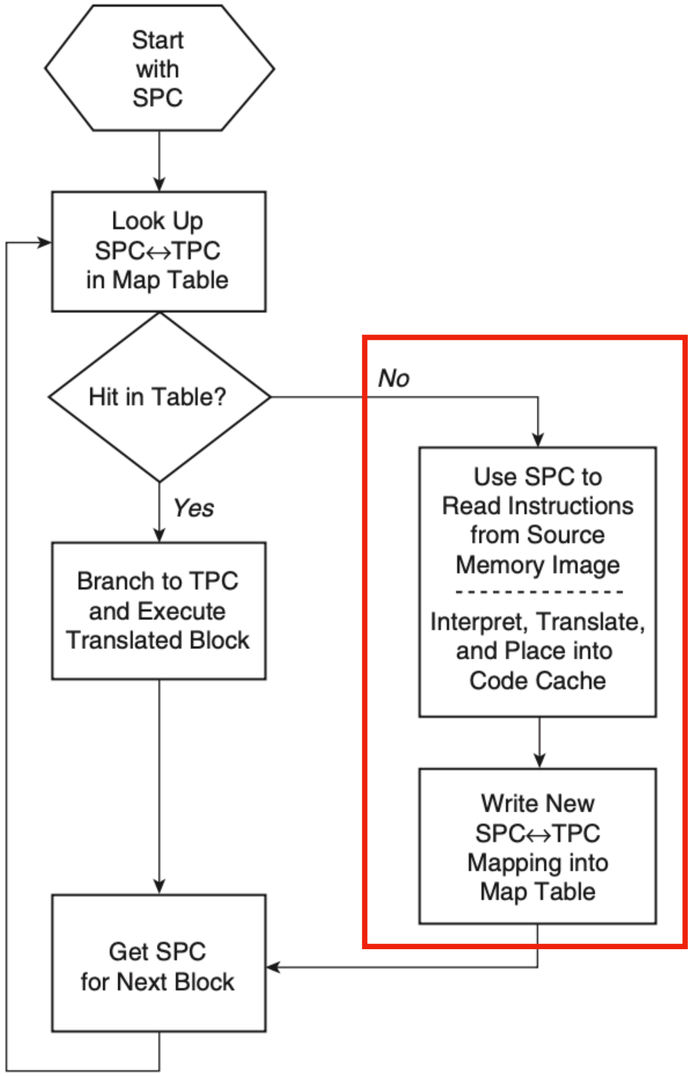
Dynamic Binary Translation Process

- Source binary is loaded into memory
- The EM follows the source program's control flow and queries the map table
- If a hit, EM executes the next block of code until the end; and look up the next SPC in map table
- If a miss, EM begins interpreting and translating the next dynamic basic block



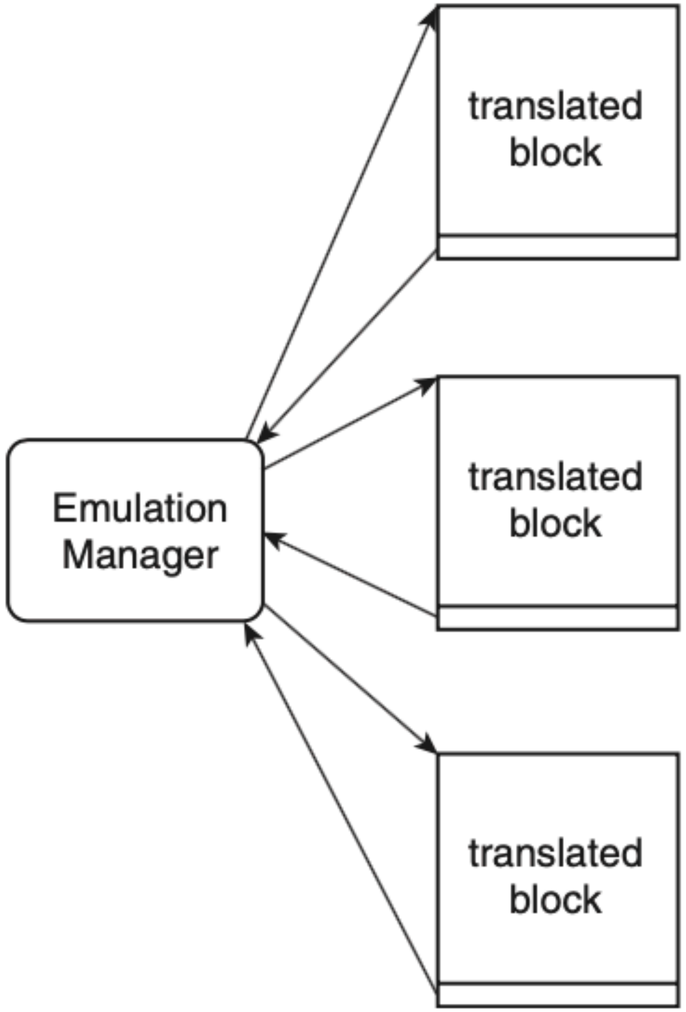
Dynamic Binary Translation Process

- EM begins interpreting the program binary either using a decode-and-dispatch or threaded method
- The interpreter dynamically generates intermediate code in the dynamic basic block (until a branch or jump is reached!)
- The interpreter calls the translator to generate target binary for instructions
- The translator puts the generated binary in the code cache



Dynamic Binary Translation Process

- Incrementally, more of the program is discovered and translated until eventually only translated code is being executed
- The EM handles control transfers from one translated block to the next

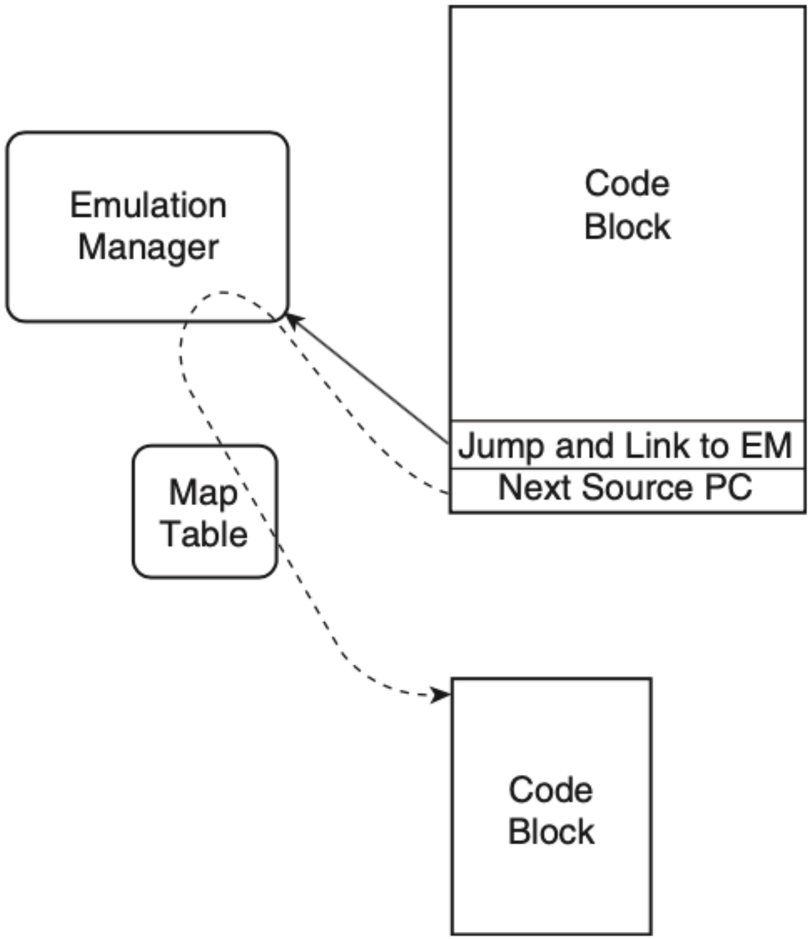


Tracking SPC

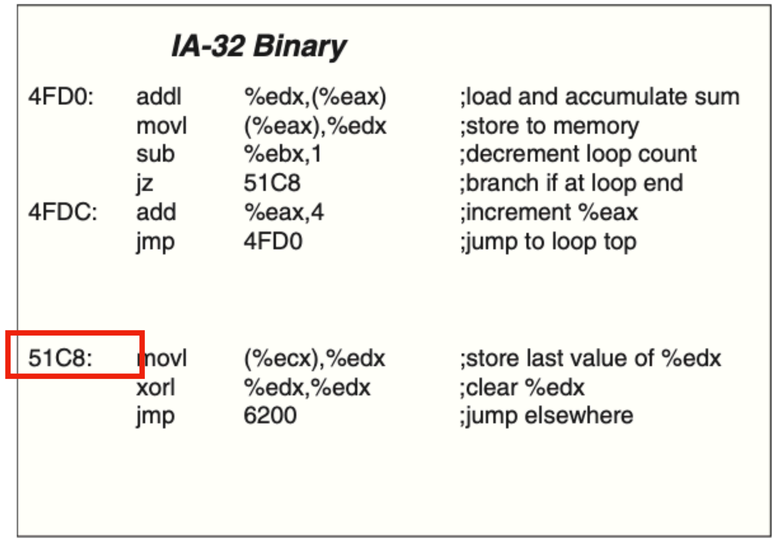
- The DBT system needs to track SPC at all times
- Control is shifted as needed between the interpreter, the EM, and translated blocks in the code cache
  - The interpreter uses SPC directly to fetch the source instructions
  - When the interpreter/translator finishes translating a basic block, it passes the next SPC to the EM before switching back

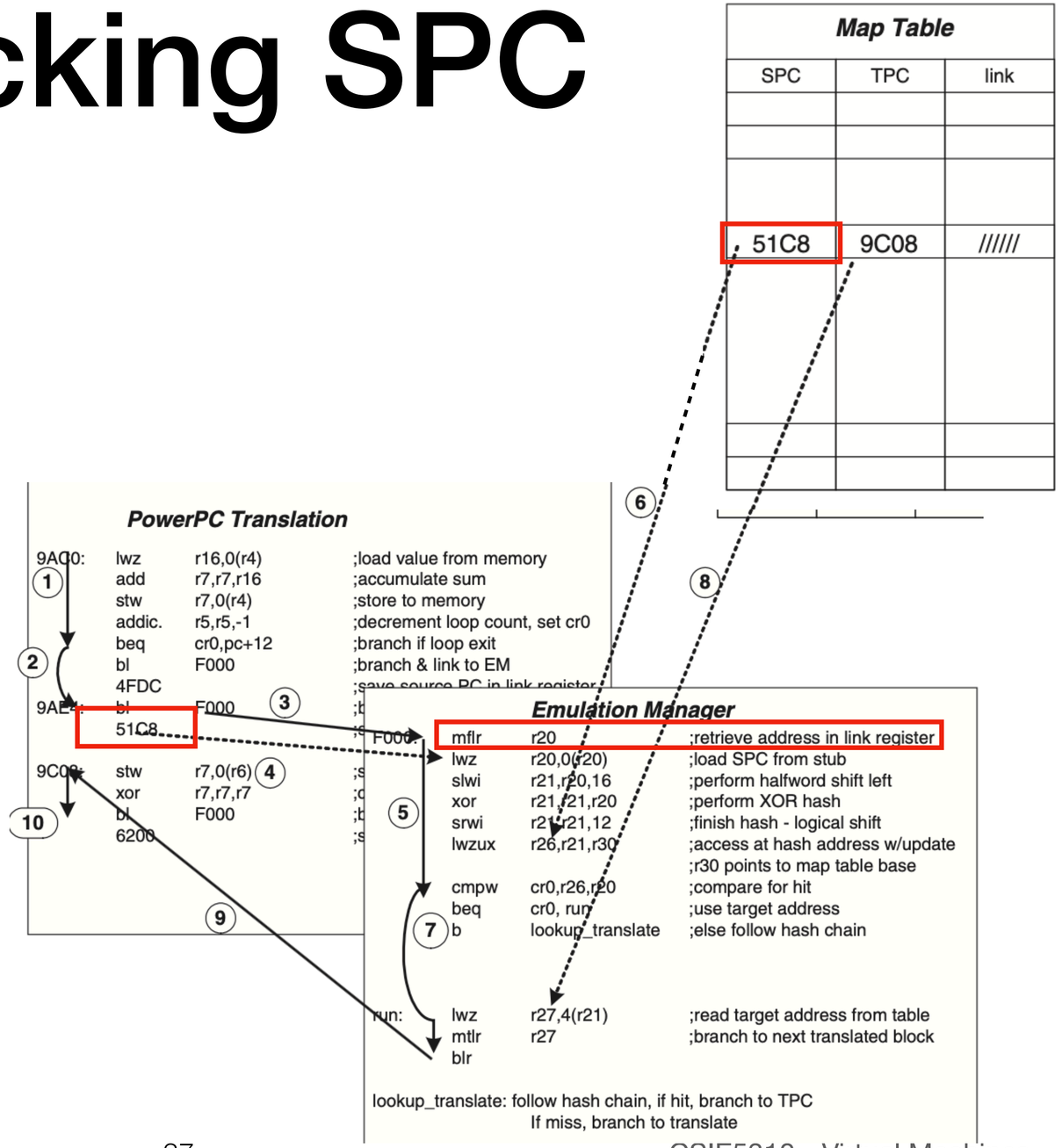
Tracking SPC

- Place the next SPC at the end of the translated block (stub)
- The translated block uses a jump and link (JAL) instruction to switch to the EM
- The EM can reference the link register to get the SPC



Tracking SPC





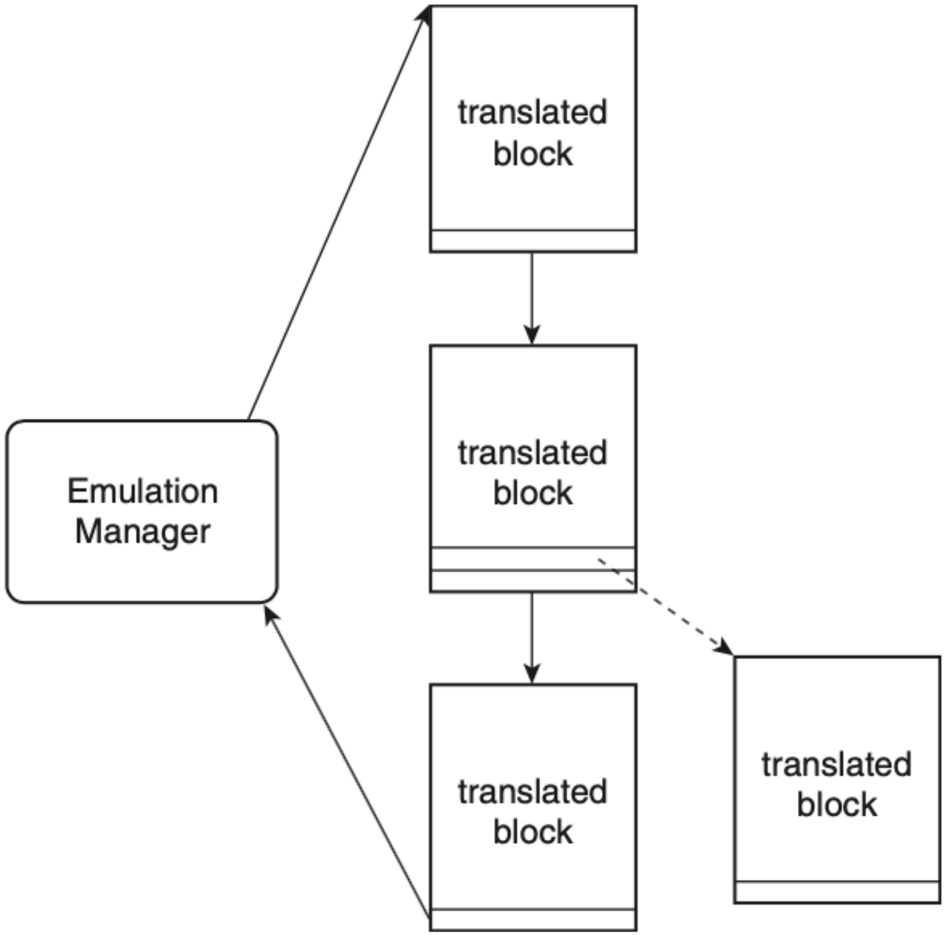
1. Translated basic block is executed.
2. Branch is taken to stub code.
3. Stub does branch and link to emulation manager entry point.
4. EM loads SPC from stub code, using link register.
5. EM hashes SPC to 16 bits and does lookup in map table.
6. EM loads SPC value from map table; comparison with stub SPC succeeds.
7. Branch to code that will transfer code back to translation.
8. Load TPC from map table.
9. Jump indirect to next translated basic block.
10. Continue execution.

Control Transfer Optimization

- Every time a translation code block finishes execution, the EM must be reentered and perform an SPC to TPC look up
- The transfers to the EM and the map table look ups cause overhead

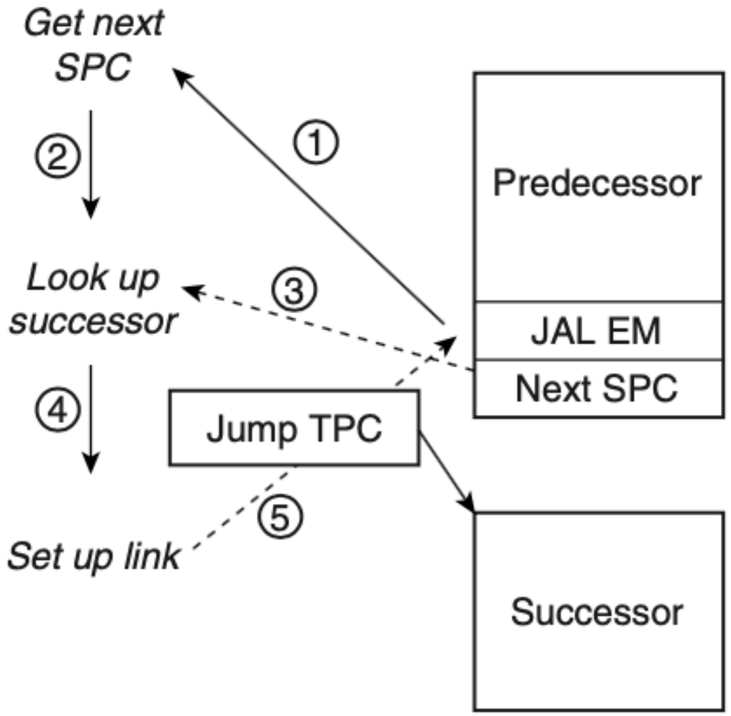
Translation Chaining

- Chaining/linking translated block to avoid branching to the EM
- Linking is done by replacing the initial jump and linking back to the EM with a direct branch to the successor translated block



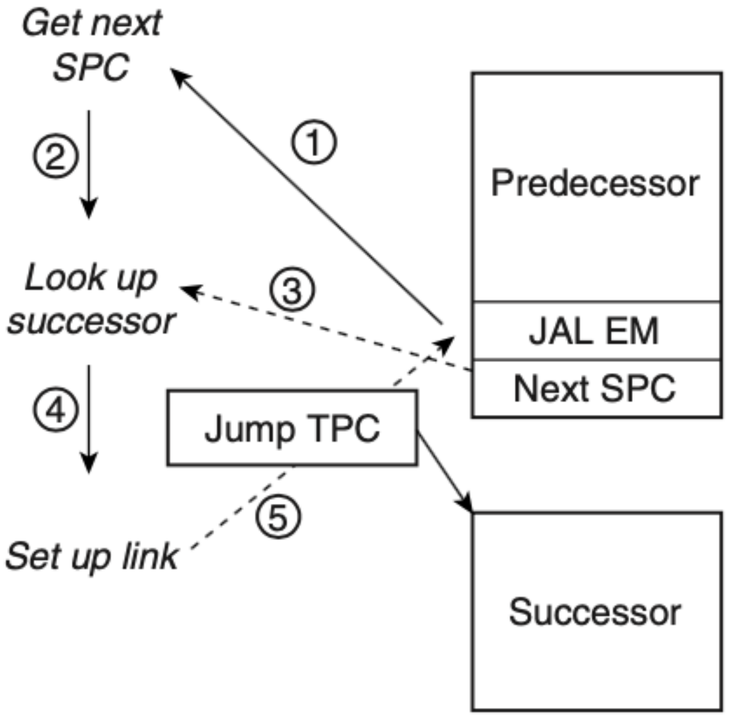
Translation Chaining

- Normal stub (JAL EM + Next SPC) is inserted into the predecessor's block if its successor block has not been translated
- When the successor block has been translated, and the predecessor block exits to the EM:
  - The EM gets the block's TPC from the map table
  - The EM overwrites the JAL instruction in the predecessor block with a direct jump to the TPC of the successor block
- Any problems?



Translation Chaining

- Problems: the scheme works only if the destination of a branch or jump never changes: direct jumps/branches
  - Cannot associate a single SPC with the translation block terminated by the jump/ branch register of an indirect jump/ branch
  - Cannot chain translated blocks for conditional jumps/branches: must handle condition codes



Software Indirect Jump Prediction

- Address the limitation in code block chaining for indirect branches/jumps
- Predicts hot jump target SPC values to avoid memory references in SPC map table look up in the EM; could couple with profiling techniques to provide accurate information about the jump targets

Summary: Binary Translation

- The translation is known to be slow, so how could binary translation be used in practice?
  - The translation is usually done once, and store the translated code in memory
  - How does binary translation compare to interpreters?
- Why static translation is not used as often as a dynamic binary translation?
  - Code discovery and code location problems
- What major optimizations are used in dynamic binary translation to make it practical?
  - Register Mapping
  - Reducing SPC/TPC table lookups and switches to the EM
    - Back patching for direct branches (translation chaining)
    - Indirect jump prediction

Agenda

- Emulation
- Interpretation
- Binary Translation
  - **More Challenges**
- Case Study of Binary Translation — QEMU

Source/Target ISA Issues

- Register Architectures
  - Challenges in register mappings; may not be always able to map
    - What about instructions that read/write system registers?
- Data Formats and Arithmetic
  - Floating point
  - Support for division vs basic shift/subtraction
- Condition codes

Source/Target ISA Issues

- Memory Address Resolution
  - Bytes vs words addressing
- Byte Order
  - Little Endian vs Big Endian
- Addressability
  - 32-bit vs 64-bit
- Atomic Instructions
  - For lock implementation

Condition Codes

- Condition codes (CCs) are special architected bits that characterize instruction results
  - Why CCs?
  - Uniformity in the way CCs are used across different ISAs:
    - PowerPC, Arm, IA-32/IA-64 have CCs; MIPS does not have CCs

Condition Codes

- In the Intel IA-32, the condition codes are a set of *flags* held in the flag register (EFLAGS); can be implicitly set by many instructions
- For example, the IA-32 integer add sets the following CCs:
  - OF: indicate integer overflow
  - SF: sign of result
  - ZF: zero result
  - AF: Carry or Borrow from bit 3 of the result
  - CF: Carry or Borrow
  - PF: Parity of the least significant bytes

Condition Codes

- The evaluation of most of the condition codes is straightforward (we'll show you an example)
- Challenging if the set of CCs between source and target machine do not fully match — CCs in target machine might be a subset of the source
  - Example: SPARC ISA has condition codes N, C, Z, V, which are equivalent to the IA-32 SF, CF, ZF, OF, but it does not have codes corresponding to AF and PF

Condition Codes

- Observations: CCs are set frequently but are seldom used
- Use *Lazy Evaluation* to optimize:
  - Instead of saving/restoring the CC or the flag, save the instruction and its operands and re-compute the CC/flag when it is needed

Condition Codes: Case Study IA-32 to PowerPC

IA-32 source code

```
        addl    %ebx,0(%eax)
        add     %ecx,%ebx
        jmp     label1
                .
                .
label1:
        jz      target
```

(a)

```
r4 ↔ %eax      IA-32 to
r5 ↔ %ebx          PowerPC
r6 ↔ %ecx              register mappings
       .
       .
r16 ↔  scratch register used by emulation code
r25 ↔  condition code operand 1      ;registers
r26 ↔  condition code operand 2      ;  used for
r27 ↔  condition code operation      ;    lazy condition code emulation
r28 ↔  jump table base address
```

PowerPC target code

addl

```
lwz     r16,0(r4)             ;perform memory load for addl
mr      r25,r16       ;save operands
mr      r26,r5                ;    and opcode for
li      r27,"addl"            ;        lazy condition code emulation
add     r5,r5,r16             ;finish addl
```

add

```
mr      r25,r6                ;save operands
mr      r26,r5                ;    and opcode for
li      r27,"add"             ;        lazy condition code emulation
add     r6,r6,r5              ;translation of add
b       label1
```

Condition Codes: Case Study IA-32 to PowerPC

IA-32 source code

```
        addl    %ebx,0(%eax)
        add     %ecx,%ebx
        jmp     label1
                .
                .
label1:
        jz      target
```

(a)

PowerPC target code

addl

```
lwz     r16,0(r4)               ;perform m
mr      r25,r16         ;save operands
mr      r26,r5                  ;     and o
li      r27,"addl"              ;         l
add     r5,r5,r16               ;finish ad
```

add

```
mr      r25,r6                  ;save opera
mr      r26,r5                  ;     and o
li      r27,"add"               ;         l
add     r6,r6,r5                ;translati
b       label1
```

```
r4 ↔ %eax       IA-32 to
r5 ↔ %ebx           PowerPC
r6 ↔ %ecx               register mappings
        .
        .
r16 ↔  scratch register used by emulation code
r25 ↔  condition code operand 1     ;registers
r26 ↔  condition code operand 2     ;  used for
r27 ↔  condition code operation     ;    lazy condition code emulation
r28 ↔  jump table base address
```

```
                .
label1:
        b1      genZF                   ;branch and link to evaluate genZF code
        beq     cr0,target              ;branch on condition flag
        .
        .
genZF
        add     r29,r28,r27             ;add "opcode" to jump table base address
        mtctr   r29                     ;copy to counter register
        bctr                            ;branch via jump table
        .
"add":
        add.    r24,r25,r26                 ;perform PowerPC add, set cr0
        blr                                 ;return
```

Agenda

- Emulation
- Interpretation
- Binary Translation
- **Case Study of Binary Translation — QEMU**

QEMU

- A generic and open-source machine emulator
- Supports both process and system VMs
  - Includes a dynamic binary translator (DBT) that supports cross-ISA emulation
  - Makes syscalls to the Linux kernel to use KVM features; runs in user space when using DBT
- Process VM
  - Supports user space emulation (ex: qemu-i386)
- System VM
  - Supports full system emulation based on either DBT or native execution
  - Integrated with KVM and Xen to provide I/O support — disabled DBT

Dynamic Binary Translation in QEMU

- QEMU includes a Tiny Code Generator (TCG) to support binary translation
- QEMU employs a two-steps translation process
  - Frontend: the TCG frontend translates guest code (in Source ISA) to TCG Intermediate Representation (IR)
  - Backend: the TCG backend translates TCG IR to host machine code (in target ISA)

Trace code: QEMU TCG

- References:
  - https://github.com/airbus-seclab/qemu_blog
  - https://github.com/airbus-seclab/qemu_blog/blob/main/tcg_p1.md
  - https://github.com/airbus-seclab/qemu_blog/blob/main/tcg_p2.md
  - https://github.com/airbus-seclab/qemu_blog/blob/main/tcg_p3.md
  - https://github.com/airbus-seclab/qemu_blog/blob/main/exec.md

Questions?