Emulation - Binary Translation

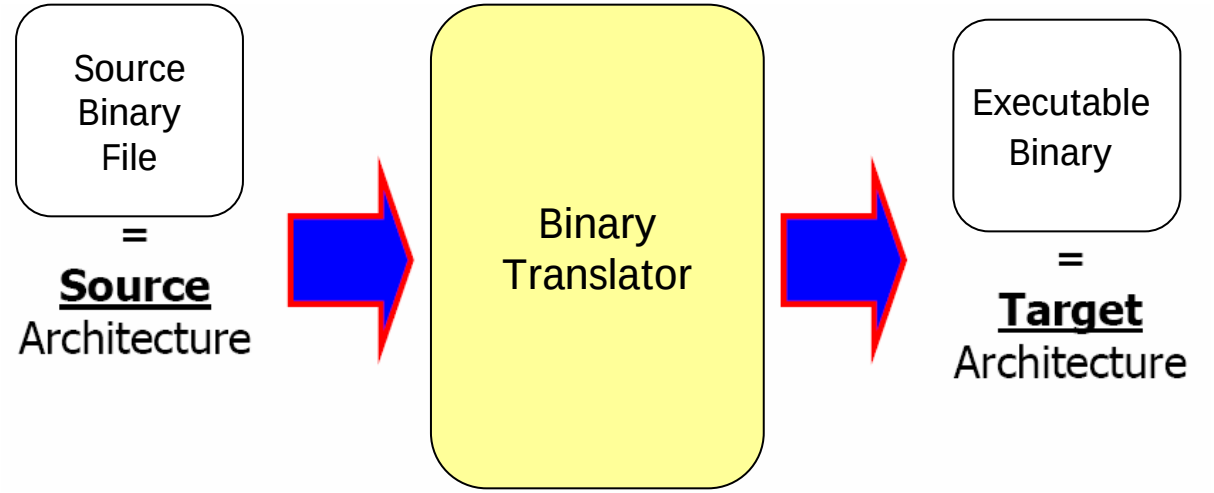
Outline

- Binary Translation
- Code Discovery and Dynamic Binary Translation
- Control Transfer Optimization
- Some Instruction Set Issues
- Summary

Binary Translation

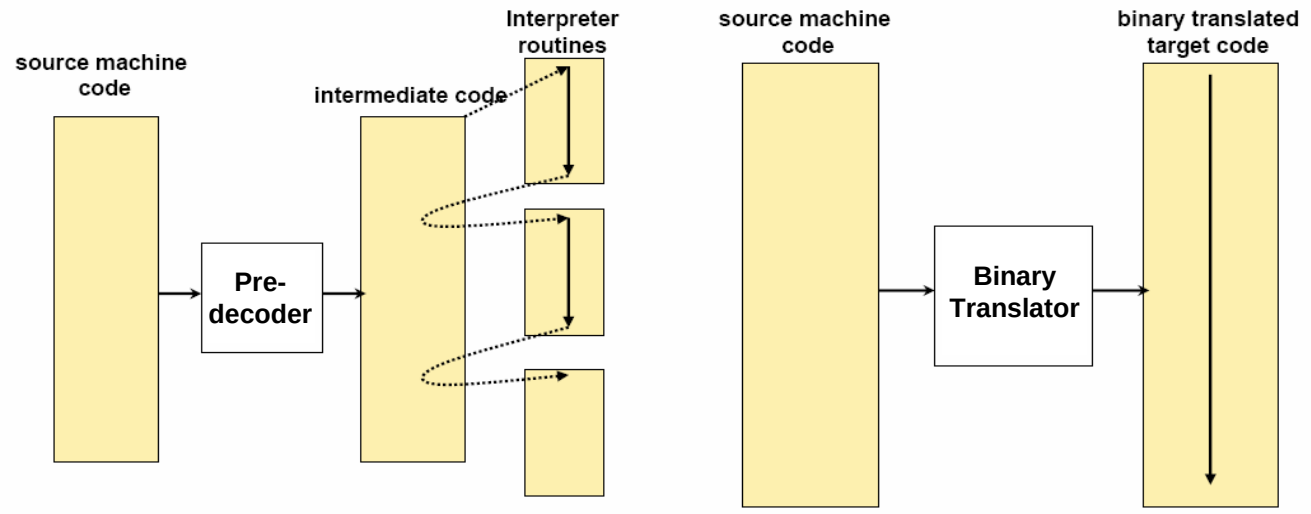
Converting a source binary program into a target binary program

- No interpretation
- Performance enhanced by mapping each individual source machine code to its customized target machine code



Comparison to Predecoding

- Similarities
  - Source machine code is converted into another form
- Differences
  - Translated code is directly executed in binary translation
  - Interpreter routines are removed



Threaded Interpretation using Intermediate code and Binary Translation

A Binary Translation Example

- X86 -> PPC binary translation
- X86 register context block
  - Architected register values for the x86 CPU are kept in a register context block in PPC's memory
  - Fetched into PowerPC register on demand
  - A pointer is maintained in r1 of PPC
- X86 memory
  - A pointer is maintained in r2 of PPC
- X86 PC
  - Is kept in r3 of PPC

A Binary Translation Example

x86 program

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

movl 4(%eax), %edx

add   %eax, 4
```

r1 points to x86-32 register context block
r2 points to x86-32 memory image
r3 contains x86-32 PC value

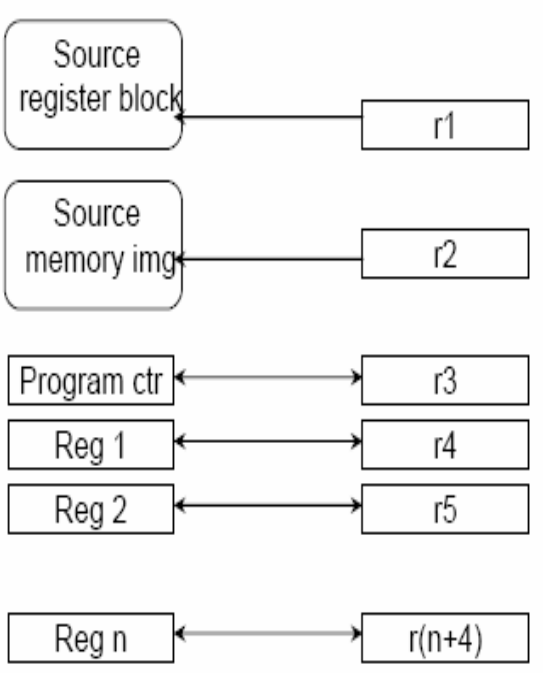
```
lwz   r4, 0(r1)        ; load %eax from register block
addi  r5, r4, 4        ; add 4 to %eax
lwzx r5, r2, r5        ; load operand from memory
lwz   r4, 12(r1)       ; load %edx from register block
add   r4, r5, r4       ; perform add
stw   r4, 12(r1)       ; put result into %edx
addi  r3, r3, 3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

lwz   r4, 0(r1)        ; load %eax again
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

lwz   r4, 0(r1)        ; load %eax again
addi  r4, r4, 4        ; add 4 to %eax
stw   r4, 0(r1)        ; place result back into %eax
addi  r3, r3, 3        ; update PC
```

State Mapping

- Mapping of more registers leads to better code
  - E.g., mapping r4 to %eax and r7 to %edx for the example



A State-Mapped Example

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

movl 4(%eax), %edx

add   %eax, 4
```

x86 program

```
r1 points to x86-32 register context block
r2 points to x86-32 memory image
r3 contains x86-32 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   r4, r5, r4     ; perform add
stw   r4, 12(r1)     ; put result into %edx
addi  r3, r3, 3      ; update PC (+3 bytes)

lwz   r4, 0(r1)      ; load %eax again
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

lwz   r4, 0(r1)      ; load %eax again
addi  r4, r4, 4      ; add 4 to %eax
stw   r4, 0(r1)      ; place result back into %eax
addi  r3, r3, 3      ; update PC
```

PPC program
Binary translated

```
r1 points to x86-32 register context block
r2 points to x86-32 memory image
r3 contains x86-32 PC value
r4 holds x86-32 register %eax
r7 holds x86-32 register %edx

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      ; add 4 to %eax
addi  r3, r3, 9      ; update PC (9 bytes)
```

PPC program
Binary translated

Code Discovery Problem

- Static binary translation
  - Can we perform binary translation ahead-of-runtime?
  - Impossible in many cases due to code discovery problem
    - How to translate indirect jump based on register value?
    - Are there always valid instructions right after jumps?
      - ✓ How to handle data intermixed with code?
      - ✓ How to handle variable-length instructions?
      - ✓ How to handle code padding?

Examples

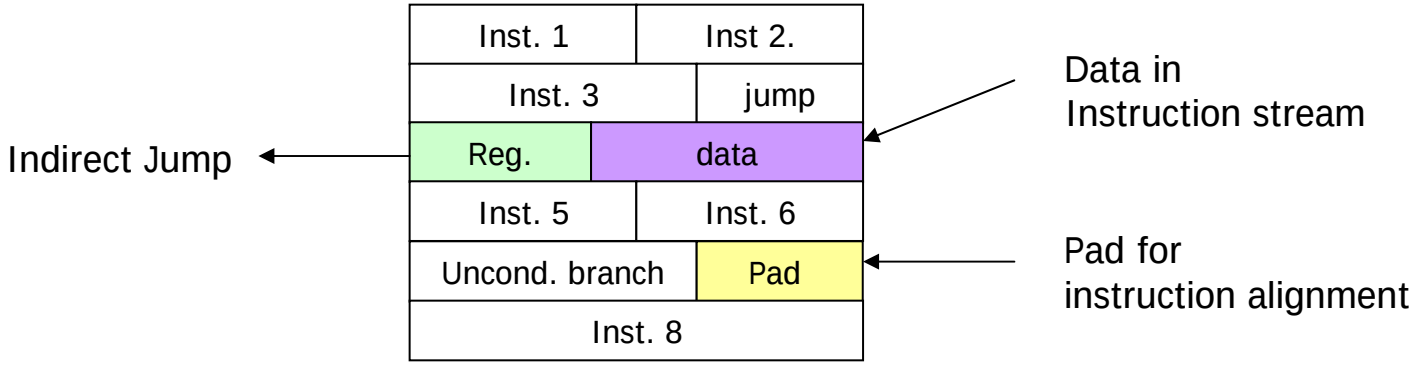
- Indirect jump based on register value
  - Occurs due to return, switch, dynamic linking, virtual call
  - The target register is not assigned until runtime
  - Impossible to determine the register content statically
- Interspersing data in code section
  - Compiler and linker do not always keep instructions and data neatly separated
- Variable length Instruction
  - CISC instruction can starts on any single byte boundary

```
                | mov %ch, 0???
31 c0 | 8b | b5 00 00 03 08 8b bd 00 00 03 00
                | movl %esi, 0x08030000(%ebp)???
```

Examples

- Pad for alignment
  - Compiler may “pad” the instruction stream with unused bytes
    - In order to align branch or jump targets on word boundary
    - Align cache line boundaries for performance reasons

- A compound example of code discovery problem



Code Location Problem

- Translated program counter (TPC) is different from architected source program counter (SPC)
- In translated code, the register content of an indirect jump would be an SPC, and only with this it is impossible to decide the next TPC
- We should map SPC and TPC addresses

x86 program

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

PPC program
Binary translated

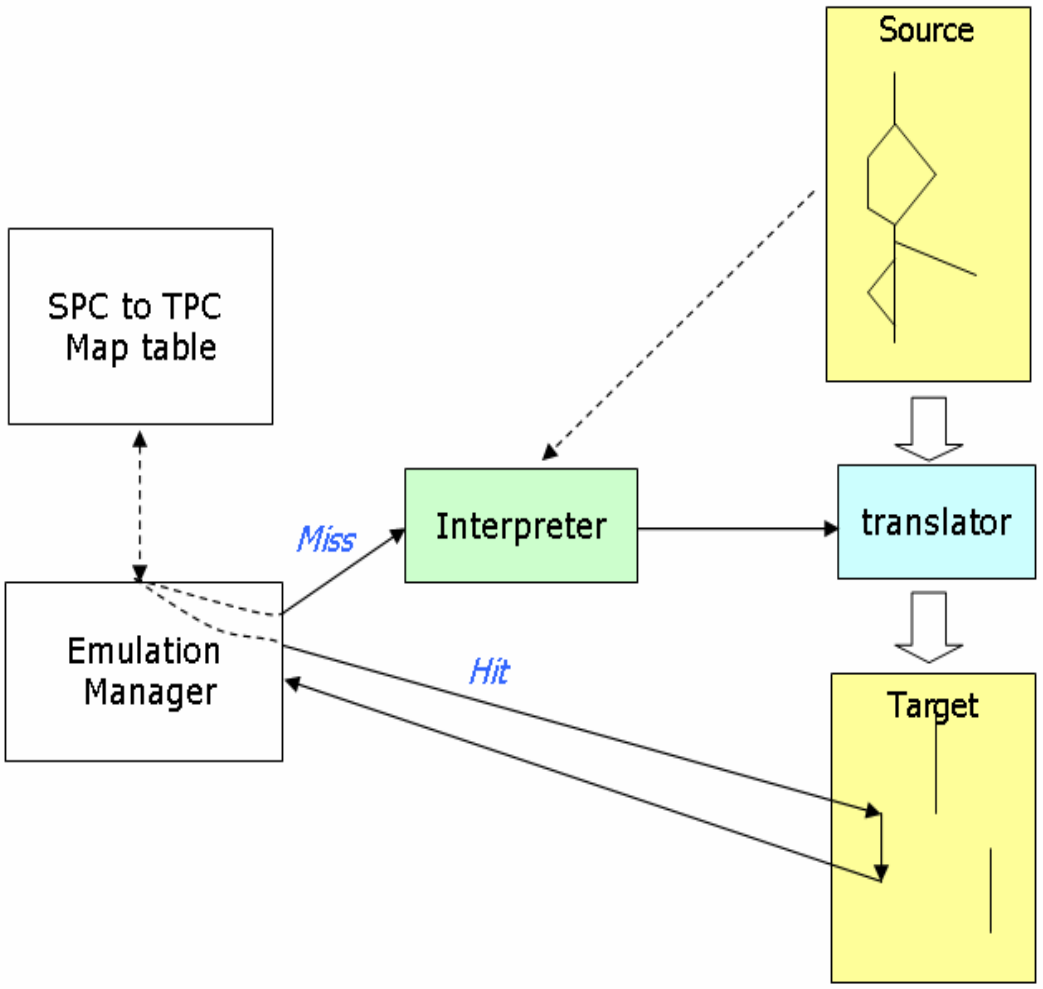
```
addi   r16, r11, 4      ; compute x86-32 address
lwzx  r4, r2, r16       ; get x86-32 jump address from x86-32 memory image
mtctr r4                ; move to count register (ctr)
bctr                    ; jump indirect through ctr
```

Dynamic Translation

Incrementally translate while program is running

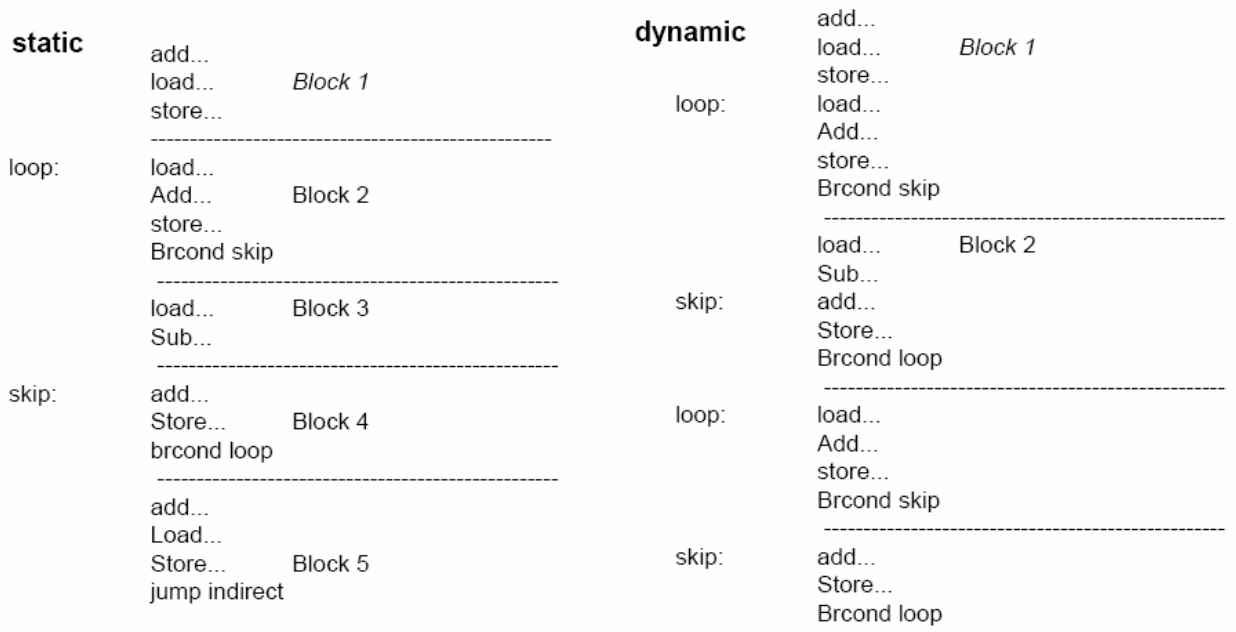
- Translate a source code block at runtime on an as-needed basis when the block gets executed
  - Works with an interpreter who provides dynamic information needed for translation (e.g., indirect jump target address)
- Place translated code into a reserved region incrementally
  - To reduce the size of memory region, typically organized as a translated code cache – holds recently used blocks only
- When a source block is to be executed, check if its translated block is in code cache, if so executes it directly
  - Use a map table from SPC to TPC when checking for code cache
- Since translation (w/interpretation) and execution are done simultaneously, so someone should coordinate them
  - Emulation manager provides a high-level control

Overview of the System



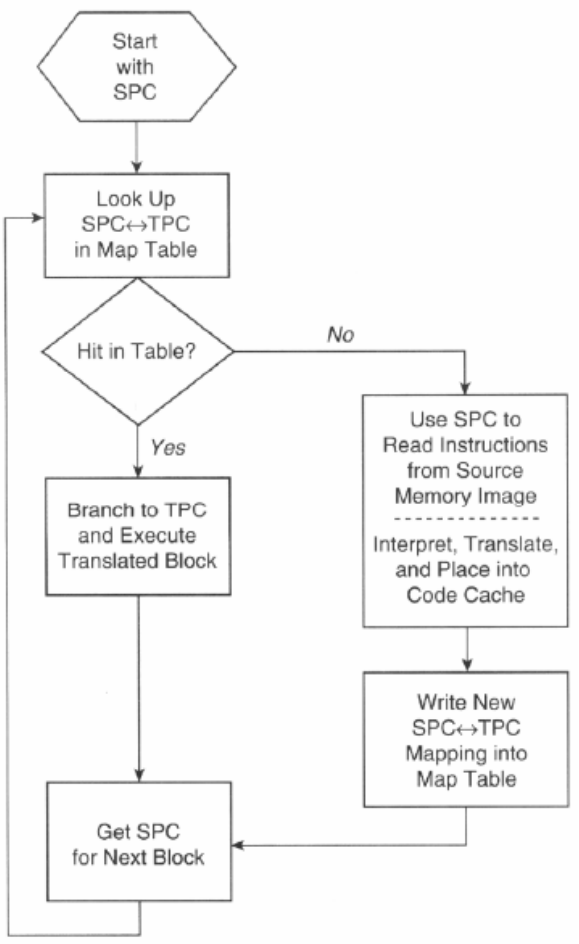
Unit of Translation

- Dynamic basic block (DBB)
  - Static basic block (SBB)
    - A single-entry and a single-exit block
  - DBB begins just after a branch, ends with next branch
  - Usually bigger then SBB
  - Translate one block at a time



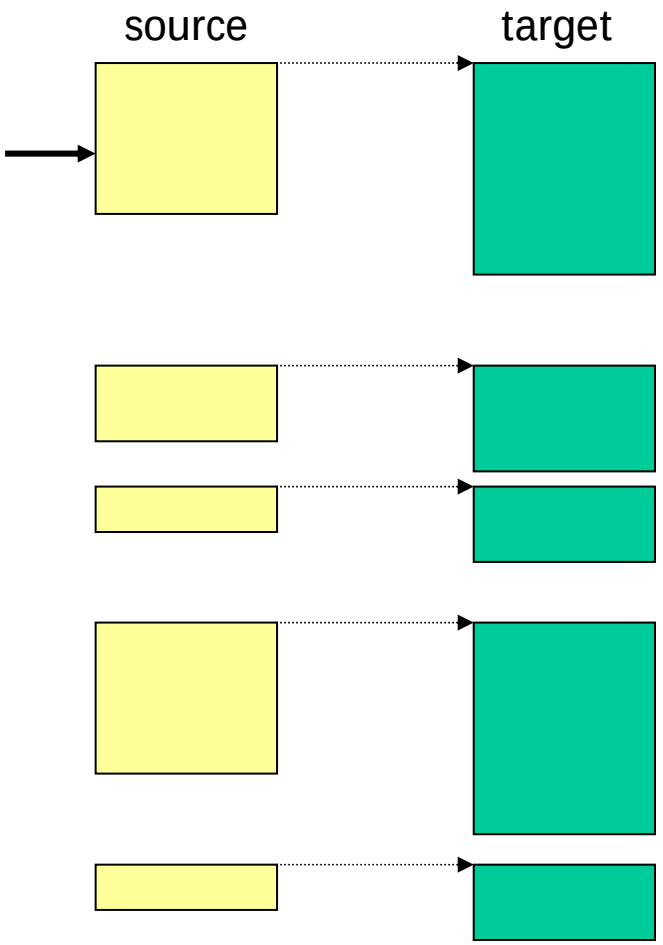
Translation Process

- EM begins interpreting the source DBB with the target DBB being newly generated
- The target DBB is placed in code cache with the corresponding SPC-TPC map included in map table
- EM now has the SPC for the next source DBB
  - Check if it is in the map table
    - Hit: execute the DBB of TPC, Miss: translate the DBB of SPC



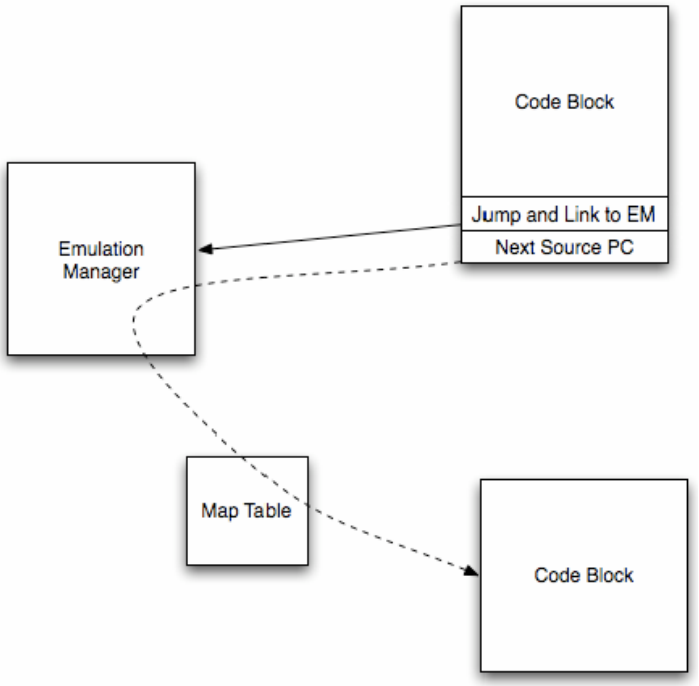
Management of DBB

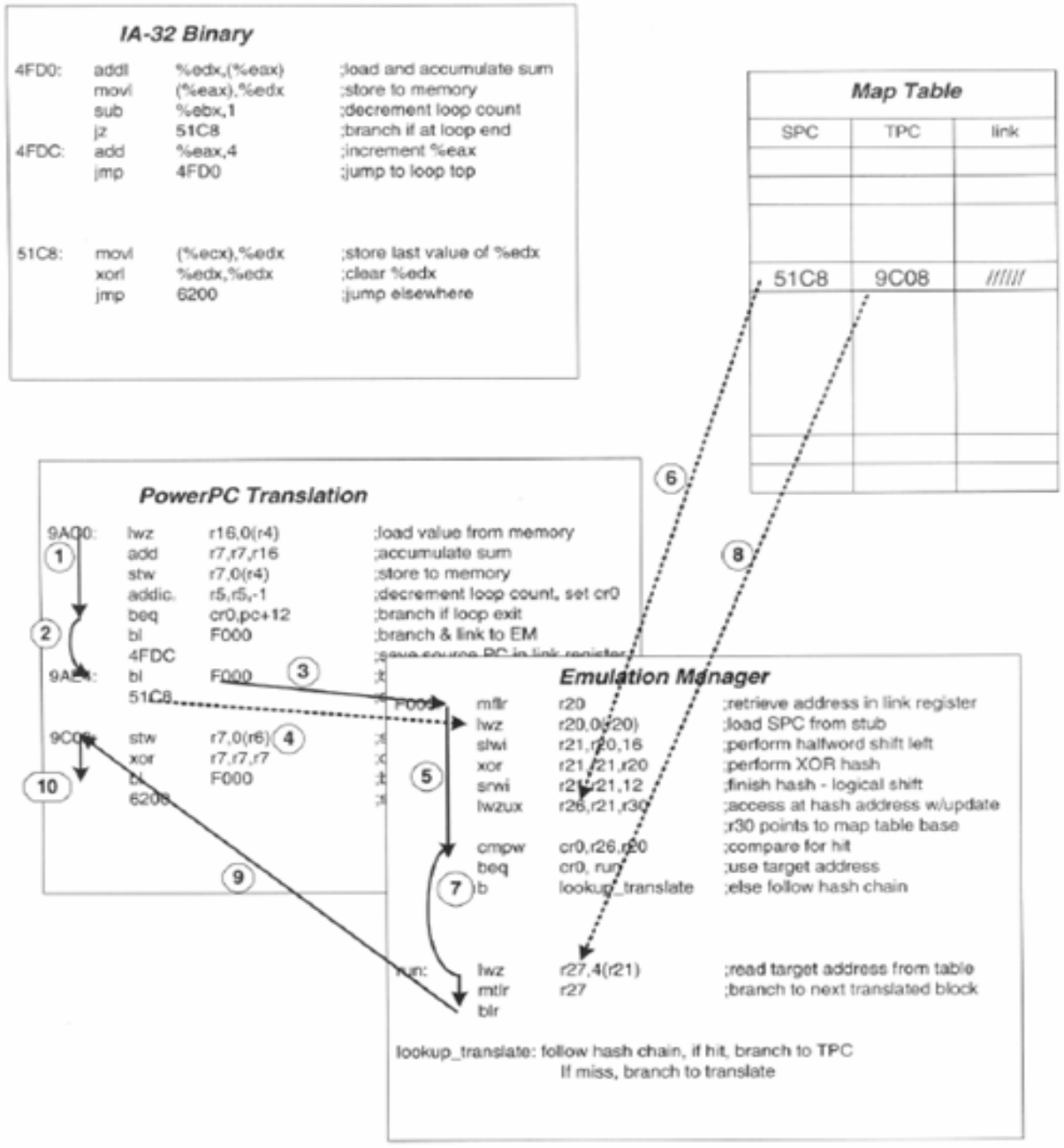
- What would happen if a branch goes to the middle of an already-translated source DBB?
  - Dividing the translated target block? Need to check ranges
  - Simply start a new translation, even if it causes duplications



Tracking the Source Program Code

- Must keep track of SPC all the time while moving between EM, interpreter, translated code DBB
  - Interpreter -> EM
    - Pass SPC to EM
  - Translated code -> EM
    - How to pass SPC to EM?
      - ✓ Map SPC to a target register
      - ✓ Save SPC at the end of DBB and use JAL (jump-and-link) instruction; EM ca get SPC via link register
- Let's see an example



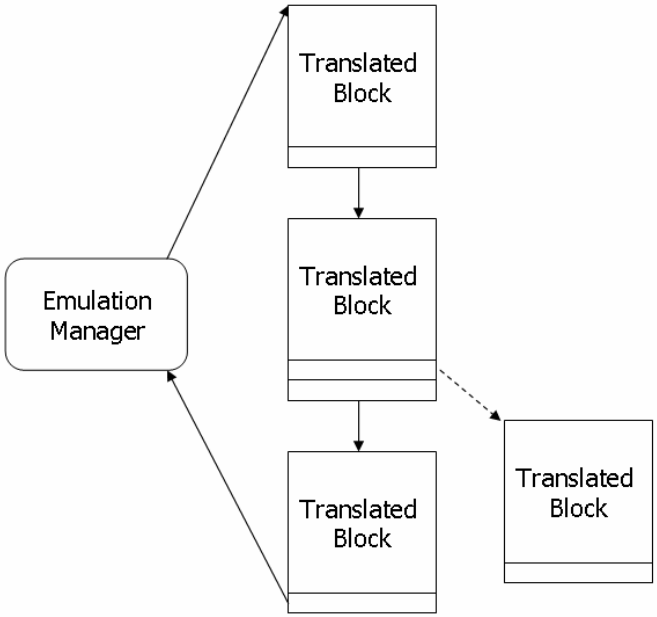


1. Translated basic block executed
2. Branch is taken to stub code
3. Stub does branch and link to EM
4. EM loads, SPC from stub code
5. EM does lookup in map table
6. EM loads SPC value from Map Table (Hit)
7. Branch to code that will transfer code back
8. Load TPC from map table
9. Jump indirect to next translated basic block
10. Continue execution

Control Transfer Optimizations

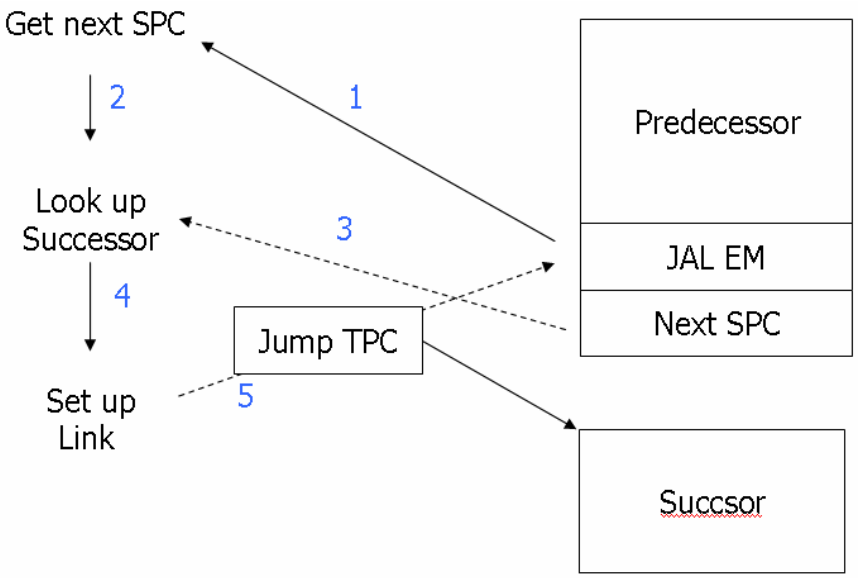
- Every time a translated DBB finishes, EM must be invoked which causes a serious overhead

- Translation Chaining
  - Equivalent to threading in interpreter
  - At the end of every translation block, blocks are directly linked together using a jump instruction
  - Dynamically done as the blocks are constructed
  - Replace the initial JAL to EM with a jump to the next translated block
  - Address of successor block is determined by mapping the SPC value to find the correct TPC



Translation Chaining

- Creating Translation Chain
  - If successor block has not been translated, the normal “stub” code (JAL to EM) is inserted
  - If it is translated, find the TPC and overwrite the previous stub with a direct jump



One Problem of Translation Chaining

- Chaining works in situation where the destination of a branch or jump never changes

- Does not work for register indirect jumps
  - Just returning to the EM works but this is expensive
  - A single SPC cannot be associated (e.g., function return)

- One solution is predicting indirect jumps
  - Based on an observation that even for indirect jumps, the actual targets seldom changes (one or two targets)

Software Indirect Jump Prediction

- **In many cases, Jump target never or seldom changes**
- **Inline caching is useful in this case**
  - Generate the following code at the end of DBB

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

  - More probable addresses obtained wirh profiling are placed at the beginning
  - Developed originally for Smalltalk
  - In our x86-> PPC example:

```
9AC0: lwz r16, 0(r4)
      add r7,r7, r16
      stw r7, 0(r4)
      addic r5, r5, -1
      beq cr0, pc+12
      bl  F000
      4FDC
9AE4: b   9C08
      51C8

9c08: stw r7, 0(r6)
      xor r7,r7,r7
      bl  F000
      6200
```

Some Complicated Issues

- **Self-modifying code**
  - A program performs store into the code area
    - E.g., modify an immediate field of an instruction in the inner loop just before entering the loop when registers are scarce
  - Translated code in the code cash no longer correspond
- **Self-referencing code**
  - A program performs load from the code area
  - Data read must correspond to the original source code, not translated version
- **Precise trap**
  - The translated code may generate an exception condition (Interrupt or trap)
  - The correct state corresponding to the original code, including SPC of the trapping instruction

Same-ISA Emulation

- Emulating an ISA on a CPU with the same ISA
  - Binary Translation is greatly simplified
  - Emulation Manager is always in control of execution
  - Execution can be monitored at any desired level of detail

- Applications
  - Simulation : collecting dynamic program properties
  - OS System call emulation : different operating systems
  - Discovery and management of sensitive privileged instructions
  - Program shepherding : watching a program execute to ensure no security holes are exploited
  - Optimize a binary program at runtime (e.g., Dynamo)

Instruction Set Issues

- Register architecture
  - Almost every ISA contains register of some kind
  - Register handling is key performance issue for emulation

- General-purpose register of target ISA
  - Holding general-purpose registers of the source ISA
  - Holding special-purpose registers of the source ISA
  - Pointing to the source register context block and the memory image
  - Holding intermediate values used by emulator

Register handling

- Case 1: # of target registers >> # of source registers
  - All source register can be mapped onto target register

- Case 2: # of target register is not enough
  - Register must be carefully handled
  - 2 registers for context block and memory image
  - 1 registers for SPC
  - 1 registers for TPC
  - Provide target registers for frequently used source register (stack pointer, condition code register, etc.)

    ※ 3 to 10 target registers are used for the above

Condition Codes

- Special architected bits
  - Characterize the instruction execution results
  - Tested by conditional branch instruction
- Condition codes vary across various ISAs
  - X86 ISA condition codes are implicitly set as a side effect of instruction execution
  - SPARC contains explicitly set codes
  - PPC has a number of condition code registers set
  - MIPS does not use any condition codes
- Emulation complexity varies depending on the use of condition codes on source and target machine

Condition Codes Emulation

- Easiest Case
  - Neither target nor source ISA use condition codes
- Almost as easy case
  - Source ISA has no condition codes, target does
    - No need to maintain any source condition state
- Difficult case
  - Target ISA has no condition codes, the source condition codes must be emulated
    - Emulation can be quite time comsuming

Most Difficult CC Emulation

- Source ISA has implicit set condition codes and target ISA does not have.
  - x86 ISA condition codes are a set of flags in EFLAGS

- x86 integer add instruction always sets six of condition flags whenever it is executed
  - OF: indicates whether an integer overflow occured
  - SF: indicates sign of result
  - ZF: indicates a zero result
  - AF: indicates carry/borrow out of bit 3 of the result (for BCD)
  - CF: indicates carry/borrow out of most significant bit of result
  - PF: indicates parity of the least significant byte of the result

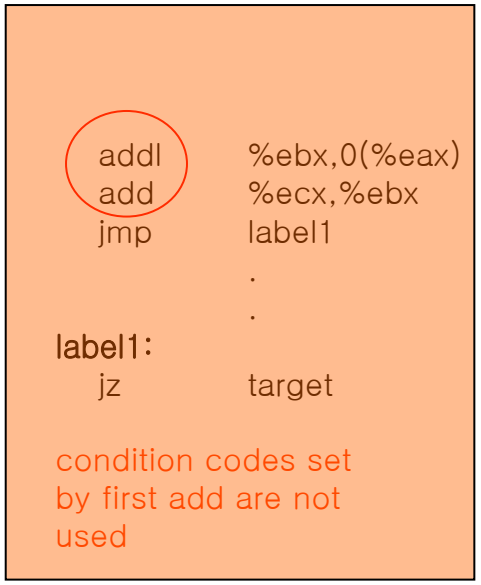
Condition Codes Emulation – x86

- Straightforward emulation
  - Every time an x86 add is emulated, each condition is evaluated
    - The SF can be determined by a simple shift
    - AF and PF requires more complex operations
    - Computing the condition code often takes more time than emulation the instruction itself!
- Condition codes are set frequently, but used rarely
  - Lazy evaluation is a good solution
    - Only operand and operation are saved (not condition code itself)
    - This allows for generation of condition codes only if they are really needed

Lazy Evaluation for x86

- Maintain a CC table that has an entry for each CC bit
  - Contains the most recent instruction who modify the bit
  - CC is evaluated when it is really used later by a branch
  - For example,
    - X86 add instruction modifies all condition code bits
    - If an add operates on two registers containing the values 2 and 3, all entries in the table will contain: add : 2 : 3 : 5
    - If a later instrution needs to test the sign bit (SF), the corresponding entry in the table is consulted, and the result field (5) is used to generate the sign bit (0)

- Another way of x86 CC emulation
  - Use a set of reserved registers instead of a table




```
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 operation

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
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
...
label1:   bl        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
          ...
"sub":    ...
"add":    add       r24,r25,2r6  ;perform PowerPC add, set cr0
          blr                    ;return
```

Condition Codes Emulation – x86

- There are still problems with condition codes
  - As already mentioned, it is possible for a trap to occur during emulation
  - In this case, the precise (source) state, including the condition code bits, must be available at the point of the trap
    - More work may be required to handle this
  - Target and source condition codes may not be entirely compatible
    - SPARC has N, C, Z, V – equivalent to x86 SF, CF, ZF, OF
    - SPARC *does not* have equivalents to AF and PF
    - Emulation can here only be simplified for some condition codes

Data Formats and Arithmetic

- Data Format Emulation
  - Most data format and arithmetic standardized
    - Integer support 2's complement
    - Floating point format uses IEEE (754) standard
  - Floating point processing may differ
    - x86 use 80 bit intermediate results, other CPU only 64
  - Emulation is possible but though it takes a longer time

- Functionality needed by source ISA not supported by target ISA
  - In all case, a simpler target ISA can build the more complex behavior from primitive operations

Memory Address Resolution

- Different ISAs can access data items of different sizes
  - One ISA supports bytes, halfword (16bit), full word (32bit)
  - Another ISA may only supports bytes, full word (32bit)
  - Multiple Instructions in the less powerful ISA can be used to emulate a single memory access instruction in a more powerful ISA
  - Most ISAs today address memory to the granularity of single bytes
    - If a target ISA does not support, many shift/and operations are required

Memory Data Alignment

- Some ISAs align memory data on “natural” boundaries
  - Word access by 00 low address bits, half word access by 0

- If an ISA does not require “natural” boundary, it is said to support unaligned data
  - One way is breaking up word accesses by byte accesses
  - Runtime analysis can reduce code expansion by finding out aligned accesses

Byte Order

- Little endian vs. big endian
  - Order bytes within a word such that the most significant byte is byte 0 (big endian) or byte 3 (little endian)
  - X86 support little endian while PPC support big endian
- It is common to maintain the guest data image in the same byte order as assumed by the source ISA, but this is not easy
  - E.g., Guest ISA's storeword performed in host ISA will save bytes in a different order
    - Problematic if there is an access to each byte which will be done in different order
  - Emulation code has to modify addresses when emulating a little-endian ISA on a big-endian machine (or vice versa)
    - A loadword should be implemented by a sequence of loadbytes
  - Some ISA support both endian orders (via mode bit)
- Byte order Issue and Operating System call
  - Guest data accessed by the host operating system has to be converted to the proper byte order

Summary

- Chapter Summary
  - Reviewed Emulation and Translation method
  - Performance tradeoff may be needed

- Performance Tradeoffs

	Memory Requirement	Start-up Performance	Steady-state Performance	Portability
DnD Interpreter	Low	Fast	Slow	Good
Indirect Threaded Interpreter	Low (higher then DnD)	Fast	Slow (slightly better then DnD)	Good
Direct Threaded Interpreter with Predecoding	High	Slow	Medium	Medium
Binary Translation	High	Slow	Fast	Poor