Dynamic binary translation

Sam King

Sensitive non-privileged instructions

- Privileged instruction
  - Trap when called from CPU user mode
- Sensitive instruction
  - Leaks information about physical state of processor
    - E.g., sidt
- Fully virtualizable processor
  - All sensitive instructions are privileged

Key insight

- Solution: simulate the OS, let user-mode code run natively
  - Simulation is flexible
    - Can interpose on all instructions
    - Problem: simulation is too slow

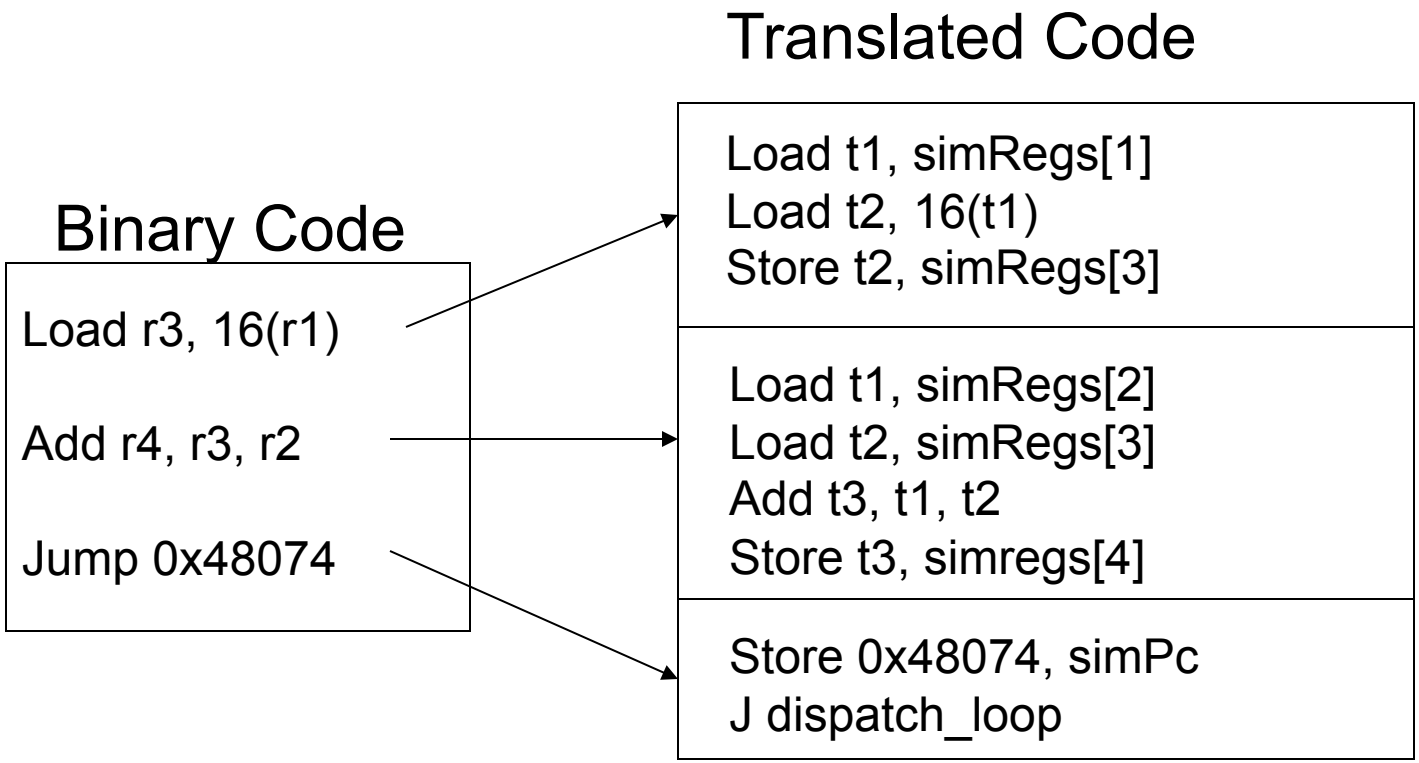
Simulation flexibility

- Normal simulations flexible, slow
- Can we simulate fast, still flexible?

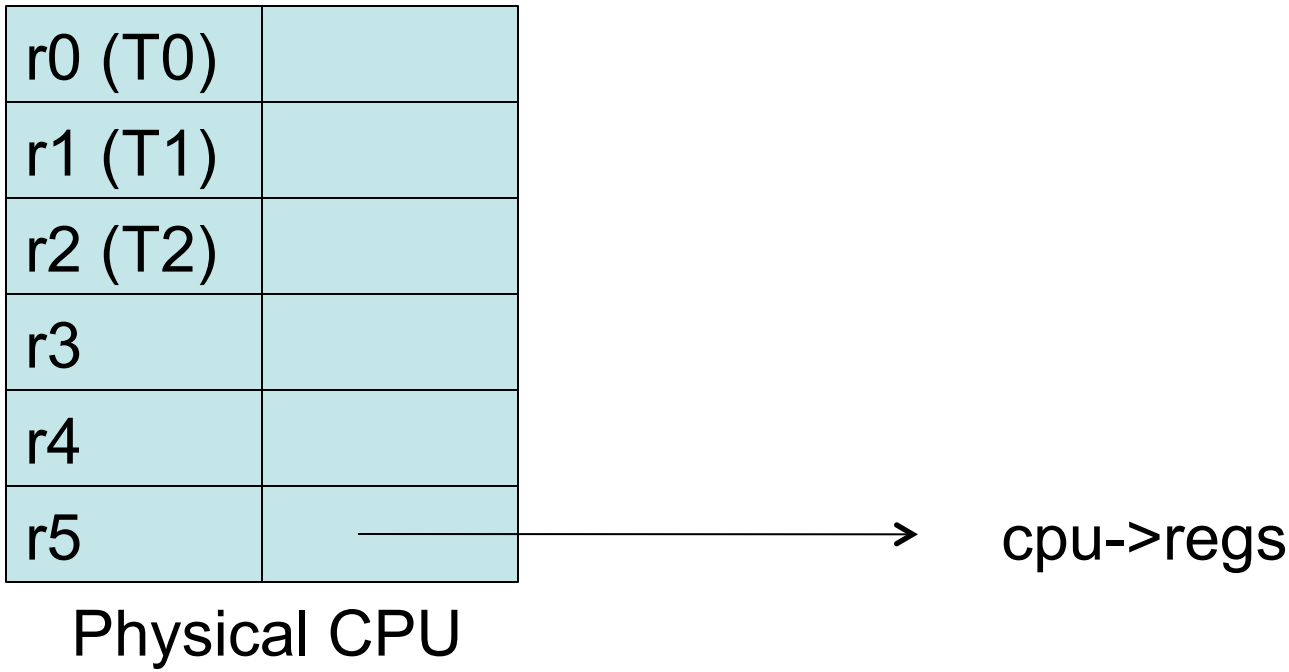
```
while(1){
        inst = mem[PC]; // fetch
        if(inst == add) { // decode
                // execute
                reg[inst.reg1] = reg[inst.reg2] +
                                reg[inst.reg3];
                PC++;
        }
}  // repeat
```

Dynamic binary translation

- Simulate basic block using host instructions.
- Cache translations

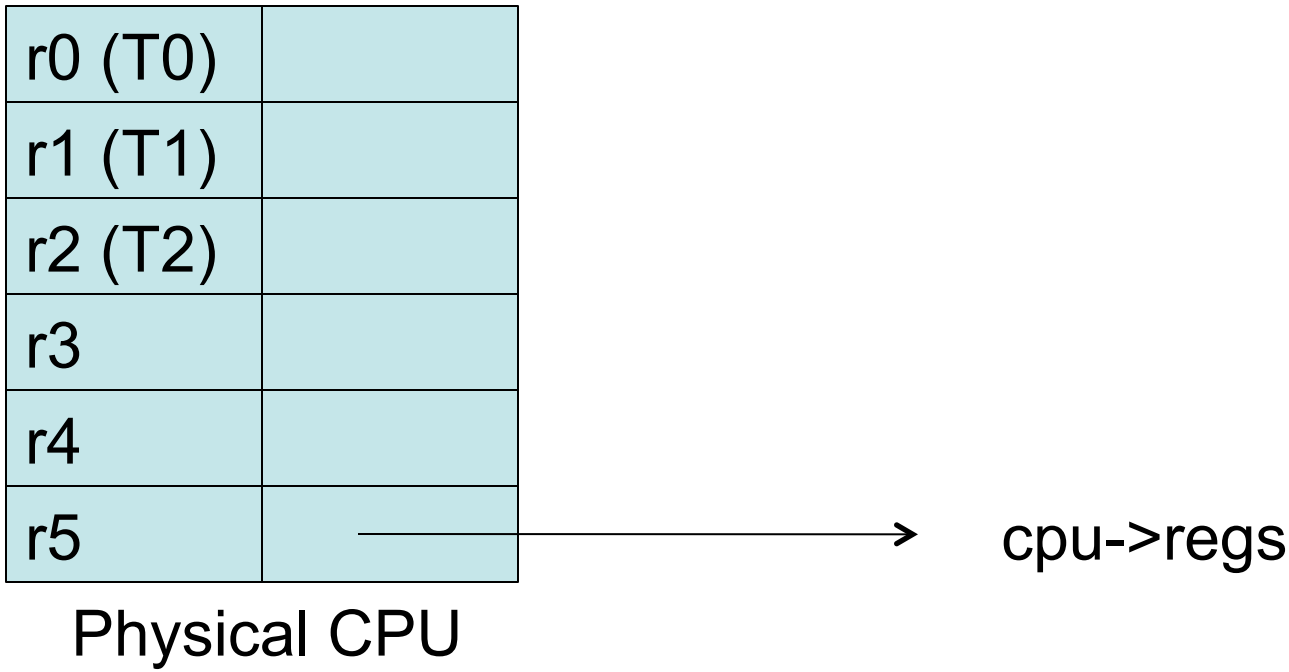


Dynamic binary translation



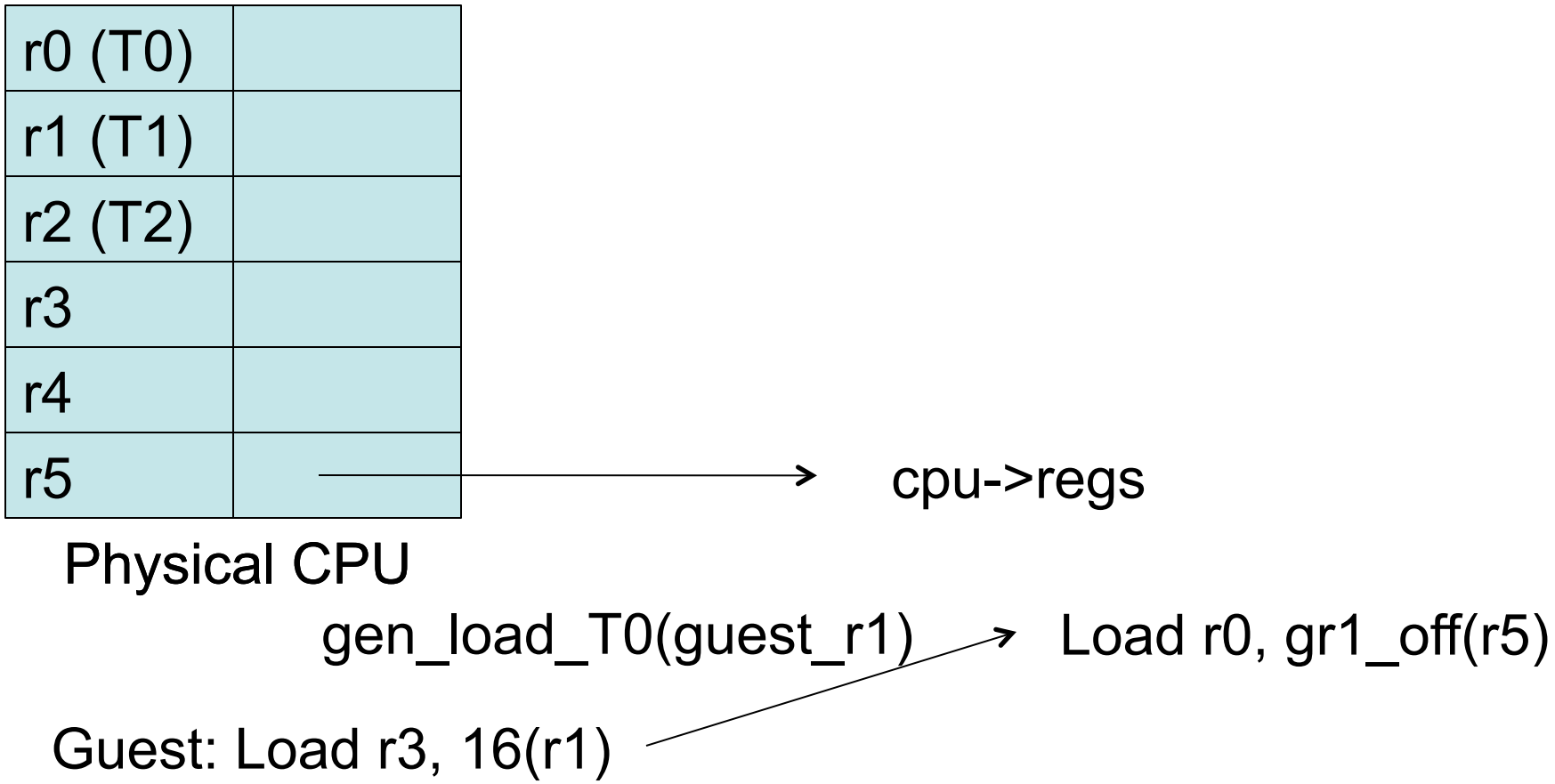
Why don’t we use a one-to-one mapping for host and guest reg?

Dynamic binary translation

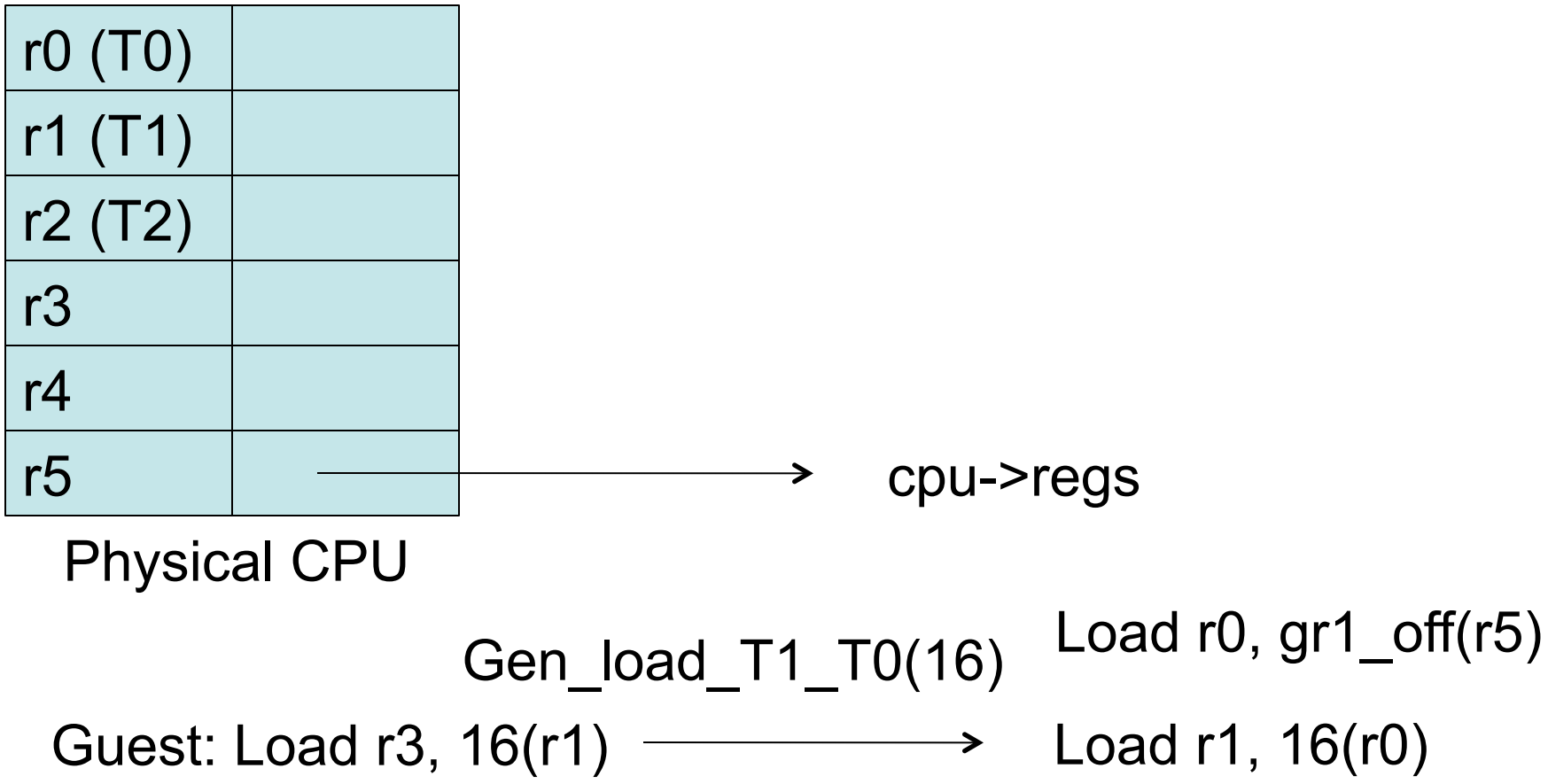


Guest: Load r3, 16(r1) → Load r1, imm(r0)

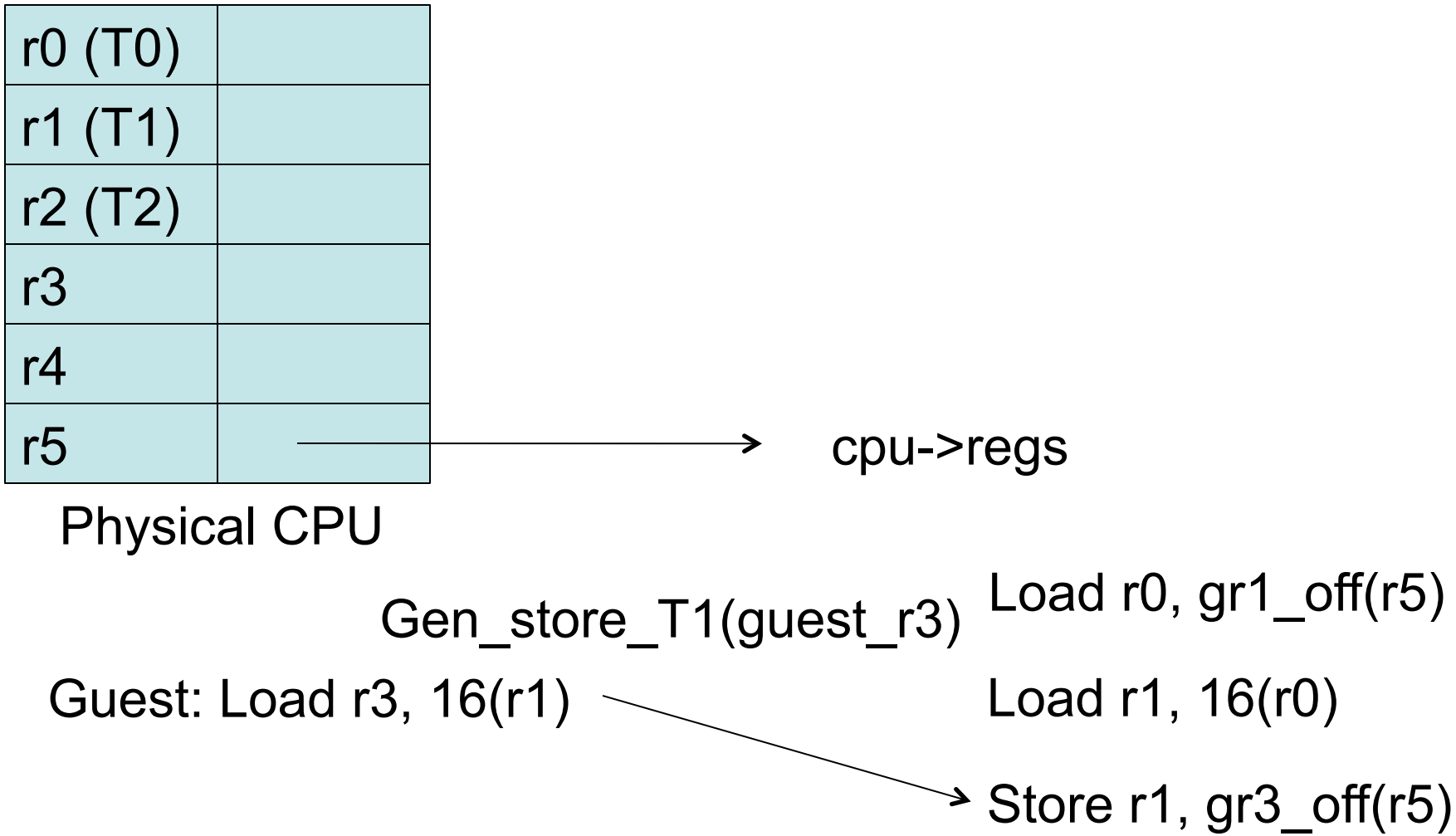
Dynamic binary translation



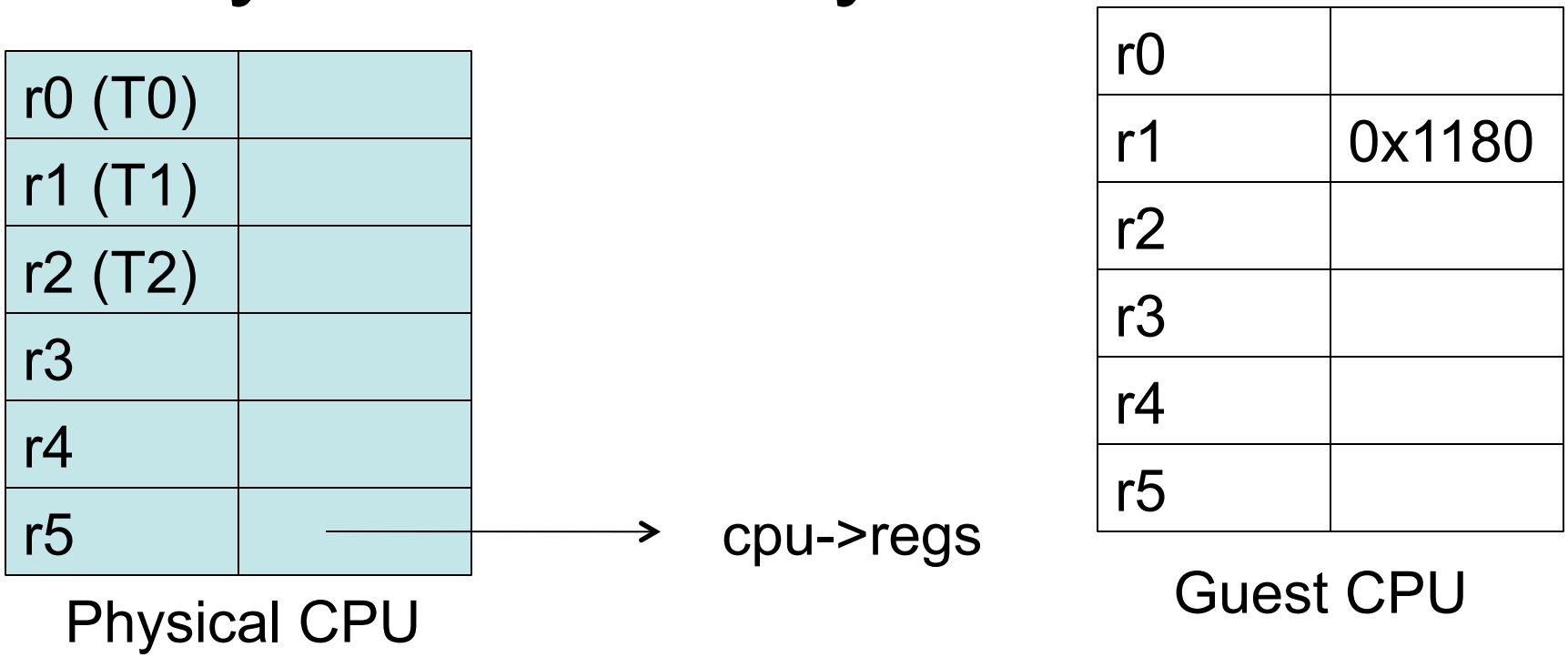
Dynamic binary translation



Dynamic binary translation



Dynamic binary translation



Guest: Load r3, 16(r1)

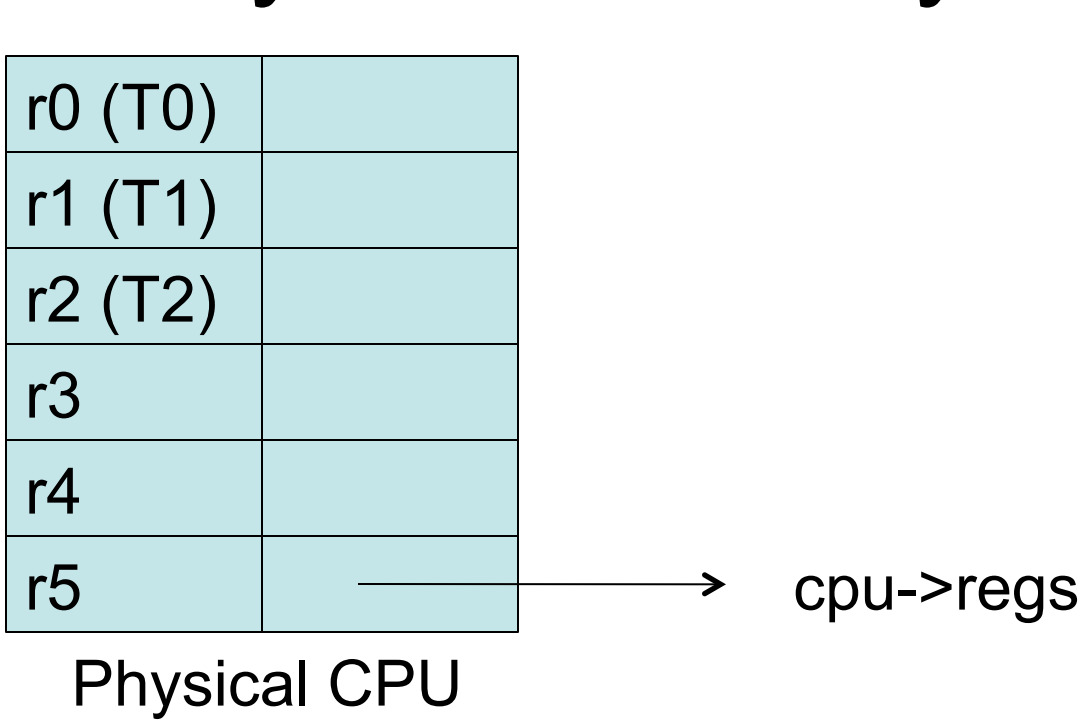
Guest mem 0x1190 -> 0x1234

Load r0, gr1_off(r5)

Load r1, 16(r0)

Store r1, gr3_off(r5)

Dynamic binary translation



r0	
r1	0x1180
r2	
r3	
r4	
r5	

Guest CPU

Load r0, gr1_off(r5)

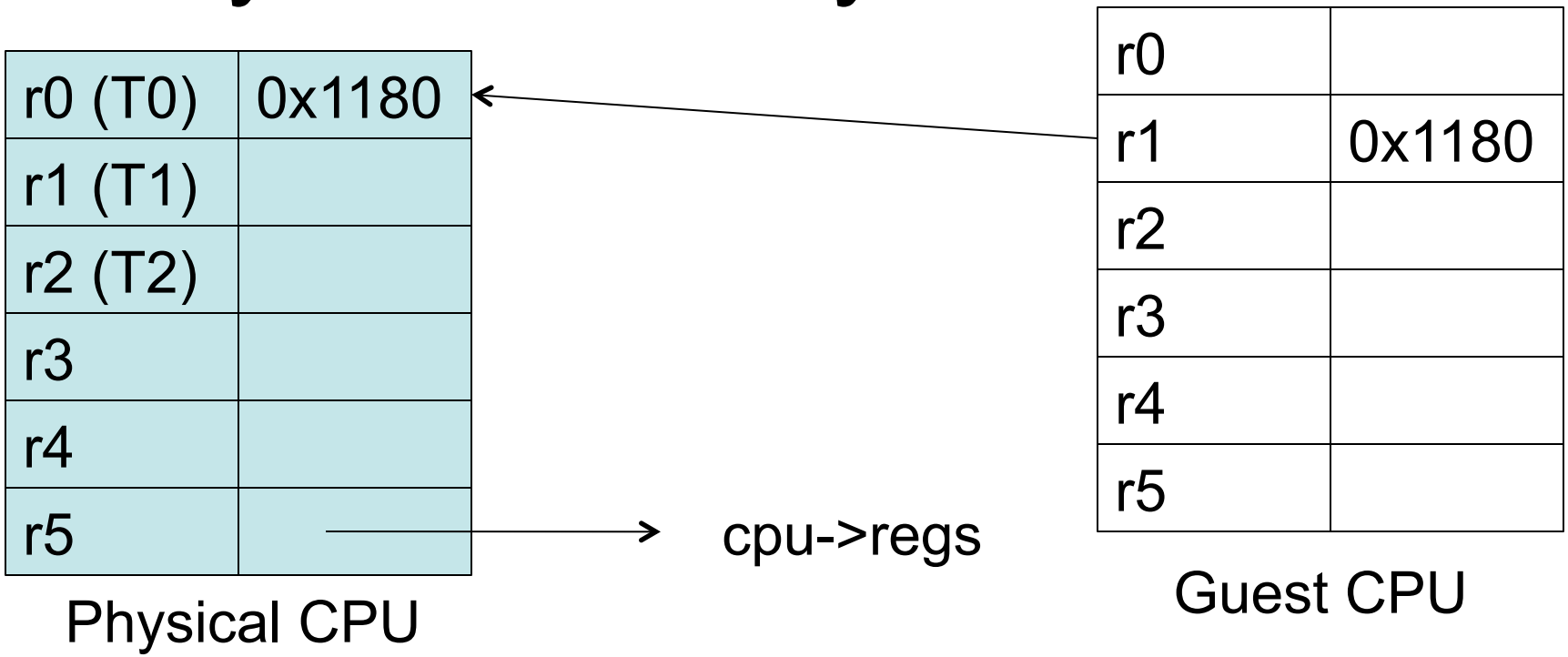
Load r1, 16(r0)

Store r1, gr3_off(r5)

Guest: Load r3, 16(r1)

Guest mem 0x1190 -> 0x1234

Dynamic binary translation



Load r0, gr1_off(r5)

Guest: Load r3, 16(r1)

Guest mem 0x1190 -> 0x1234

Load r1, 16(r0)

Store r1, gr3_off(r5)

Dynamic binary translation

r0 (T0)	0x1180
r1 (T1)	
r2 (T2)	
r3	
r4	
r5	

Physical CPU

r0	
r1	0x1180
r2	
r3	
r4	
r5	

Guest CPU

Guest: Load r3, 16(r1)

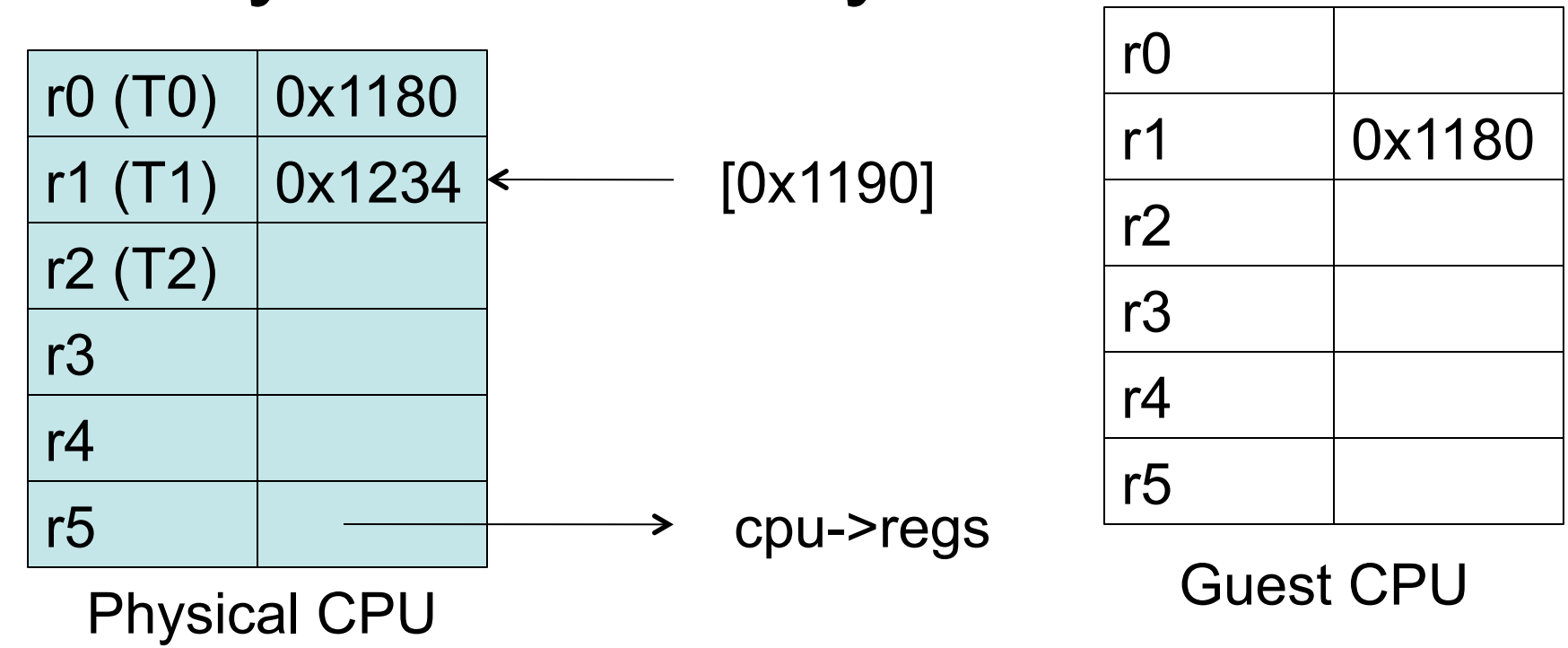
Guest mem 0x1190 -> 0x1234

Load r0, gr1_off(r5)

Load r1, 16(r0)

Store r1, gr3_off(r5)

Dynamic binary translation



Load r0, gr1_off(r5)

Load r1, 16(r0)

Store r1, gr3_off(r5)

Guest: Load r3, 16(r1)

Guest mem 0x1190 -> 0x1234

Dynamic binary translation

r0 (T0)	0x1180
r1 (T1)	0x1234
r2 (T2)	
r3	
r4	
r5	

Physical CPU

cpu->regs

r0	
r1	0x1180
r2	
r3	
r4	
r5	

Guest CPU

Guest: Load r3, 16(r1)

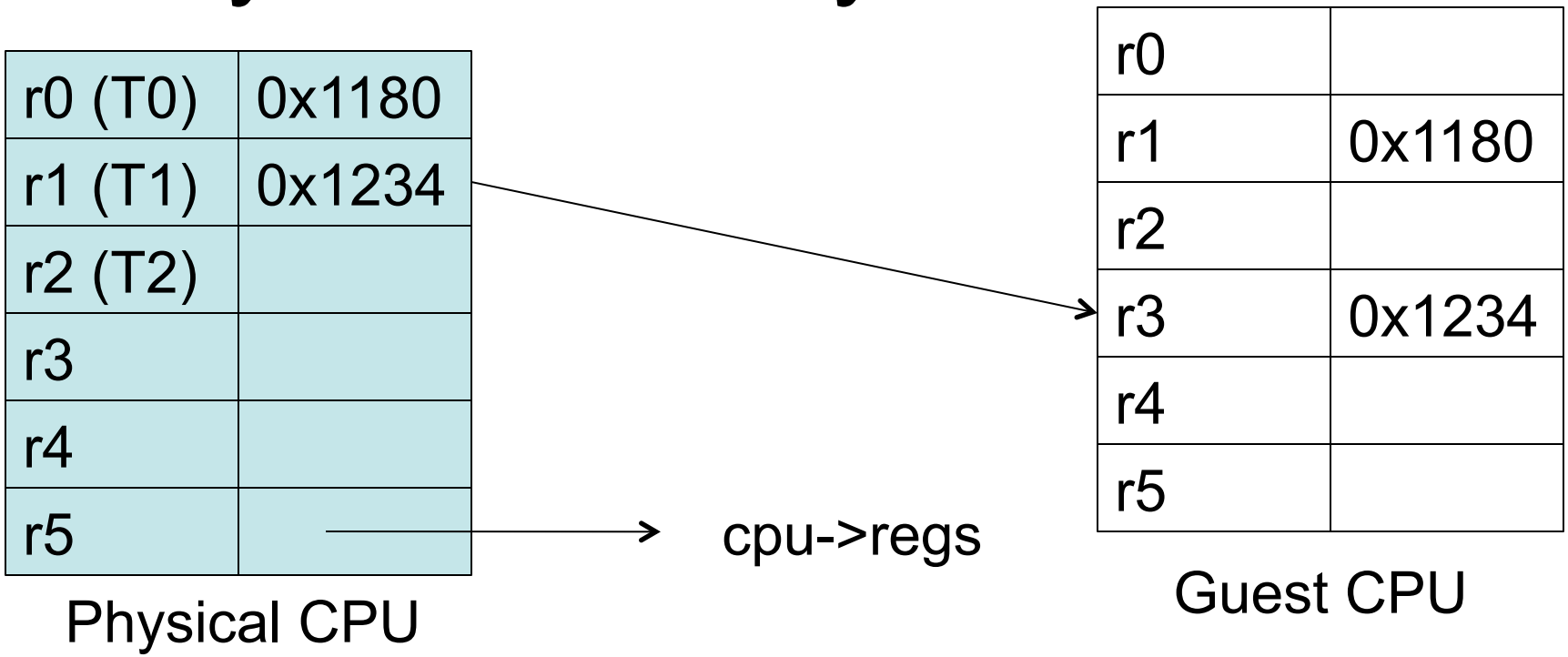
Guest mem 0x1190 -> 0x1234

Load r0, gr1_off(r5)

Load r1, 16(r0)

Store r1, gr3_off(r5)

Dynamic binary translation



Guest: Load r3, 16(r1)

Guest mem 0x1190 -> 0x1234

Load r0, gr1_off(r5)

Load r1, 16(r0)

Store r1, gr3_off(r5)

Dynamic binary translation

r0 (T0)	0x1180
r1 (T1)	0x1234
r2 (T2)	
r3	
r4	
r5	

Physical CPU

r0	
r1	0x1180
r2	
r3	0x1234
r4	
r5	

Guest CPU

Guest: Load r3, 16(r1)

Guest mem 0x1190 -> 0x1234

Load r0, gr1_off(r5)

Load r1, 16(r0)

Store r1, gr3_off(r5)

Group problem

- Generate the translation code for bitwise OR of guest code
- E.g., Or r4, r2, r1 -> r4 = r2 | r1
- 1) Write any prototypes for code gen
  - E.g., gen_load_T1_T0(imm)
- 2) Write the translation code
- Note: feel free to reuse any codegen from example

Useful functions

- Instruction: or r4, r2, r1
  - r4 = r2 | r1

- Gen_load_T0(guest reg no)
- Gen_load_T1(guest reg no)
- Gen_store_T2(guest reg no)

Solution

Performance

```
while(1){
    inst = mem[PC]; // fetch
      // decode
      if(inst == add) {
          // execute
          …
      }
}  // repeat
```

```
while(1){
    if(!translated(pc)) {
        translate(pc);
    }
    jump to pc2tc(pc);
}  // repeat
```

Dynamic binary translation for x86

- Goal: translate this basic block
- Note: x86 assembly dst value is last

```
Mov %rsp, %rbp
Sub $0x10, %rsp
Movq $0x100000, 0xfffffffffffffff8(%rbp)
```

Inst-by-inst simulation: fetch

```
void fetch(struct CPU *cpu, struct control_signals *control) {
    memset(control, 0, sizeof(struct control_signals));
    control->orig_rip = cpu->rip;
    control->opcode[control->numOpcode++] = fetch_byte(cpu);

    if(IS_REX_PREFIX(control->opcode[0])) {
        control->rex_prefix =
            control->opcode[control->numOpcode-1];
        control->rex_W = REX_PREFIX_W(control->rex_prefix);
        control->opcode[control->numOpcode-1] =
            fetch_byte(cpu);
    }
// and more of this for multi-byte opcodes
```

Inst-by-inst simulation: decode

```
If(HAS_MODRM(control->opcode[0]) {
    control->modRM = fetch_byte(cpu);
    control->mod = MODRM_TO_MOD(control->modRM);
    control->rm = MODRM_TO_RM(control->modRM);
    control->reg = MODRM_TO_R(control->modRM);

    // now calculate m addresses
    if(IS_SIB(control)){
        // more address lookup and fetching
    } else if(IS_DISP32) {
       // fetch immediate
    } else {
       addr = getRegValue(cpu, control)
    }

    // now fetch any disp8 or disp32 for addr
```

Dynamic translation

- Amortize the cost of fetch and decode
- Two different contexts
  - Translator context – fetch and decode
  - Guest context – execute

Dynamic translation for x86

- Map guest registers into these host registers

- T0 -> rax
- T1 -> rbx
- T2 -> rcx – use for host addresses

- Rdi -> holds a pointer to cpu

Inst-by-inst: `Mov %rsp, %rbp`

```
Cpu->reg[control->rm] =
    cpu->reg[control->reg];
```

```
Where rm == rbp index and reg == rsp
  index
```

DBT: Mov %rsp, %rbp

```
void mov(struct CPU *cpu, struct control_signals
   *control) {
    gen_load_T0(cpu->tcb,
                getRegOffset(cpu, control->reg));
    gen_store_T0(cpu->tcb,
                getRegOffset(cpu, control->rm));
}
```

DBT: Mov %rsp, %rbp

```
void gen_load_T0(struct tcb *tcb, uint32_t offset) {
    uint64_t opcode = offset;
    // create movq offset(%rdi), %rax
    opcode <<= 24;
    opcode |= 0x878b48;

    // store in cache
    memcpy(tcb->buffer+tcb->bytesInCache,
           translation, size);
    tcb->bytesInCache += size;
}
```

DBT: compile

```
// Mov %rsp, %rbp
Gen_load_T0(control->reg)
Gen_store_T0(control->rm)

//Sub $0x10, %rsp
gen_load_rm32_T0(cpu, control)
Gen_alu_imm32_T0(tcb, imm, SUB_ALU_OP)
Gen_store_rm32_T0(cpu, control)

// movq $0x100000, -8(%rbp)
Gen_mov_imm32_T0(cpu, control->imm)
Gen_store_rm64_T0(cpu, control)
```

Switching to guest context

```
Jump_to_tc(cpu) {
  // will return a pointer to beginning of TC buf
  void (*foo)(struct CPU *) = lookup_tc(cpu->rip)
  foo(cpu);
}
```

- What does the preamble for your TCB look like?
- How do we return back to the host (or translator) context?

DBT optimizations

```
Mov %rsp, %rbp
Sub $0x10, %rsp
movq $0x100000, -8(%rbp)
```

- Are there opportunities for optimizations?

Discussion questions

- Could you run this raw code on the CPU directly?
  - What assumptions do you have to make?
- When do you have to invalidate your translation cache

- Break into groups to discuss this

- Read embra for discussion about using guest physical vs guest virtual pc values to index tb