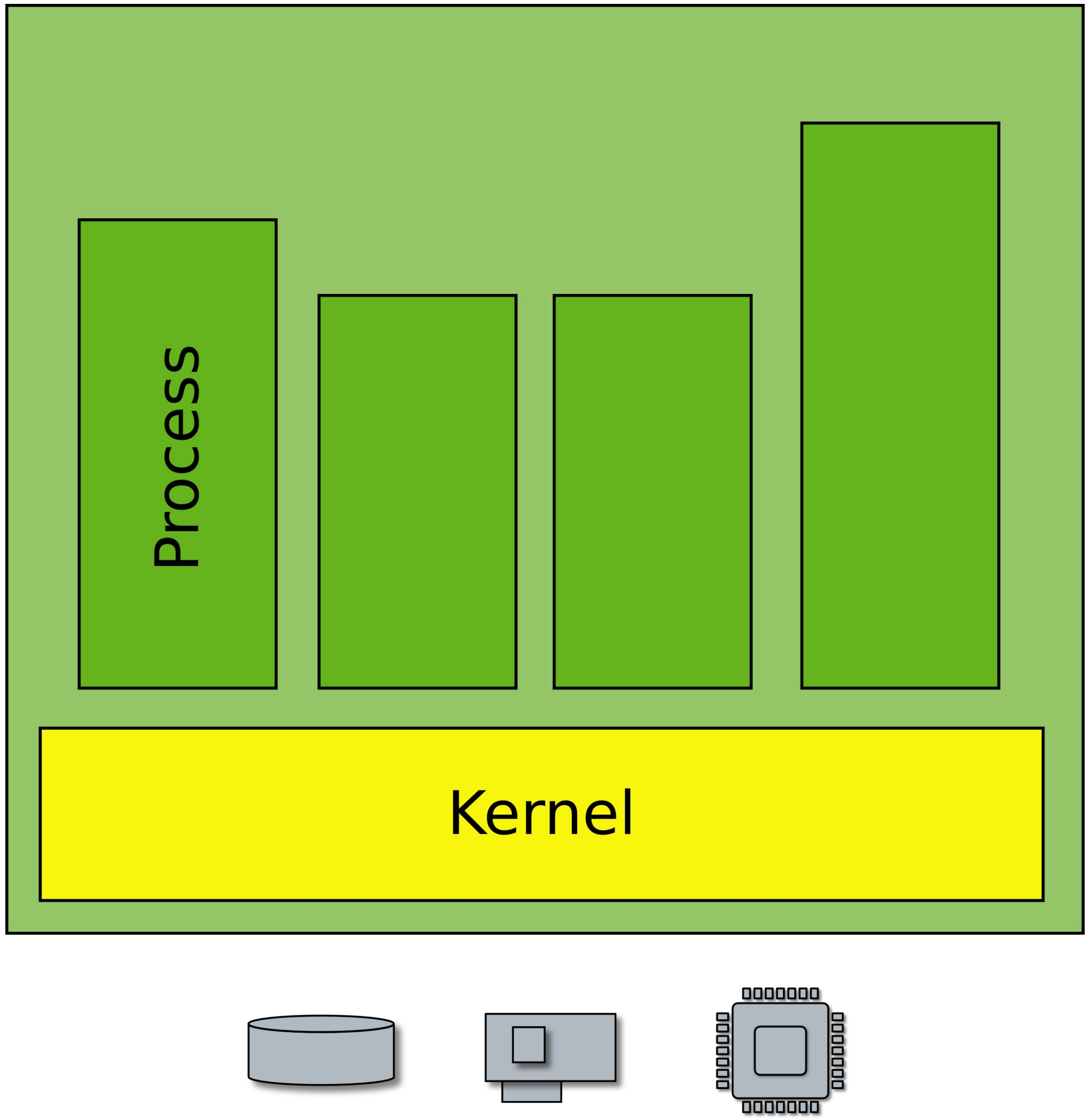
CS5460: Operating Systems

Lecture: Virtualization

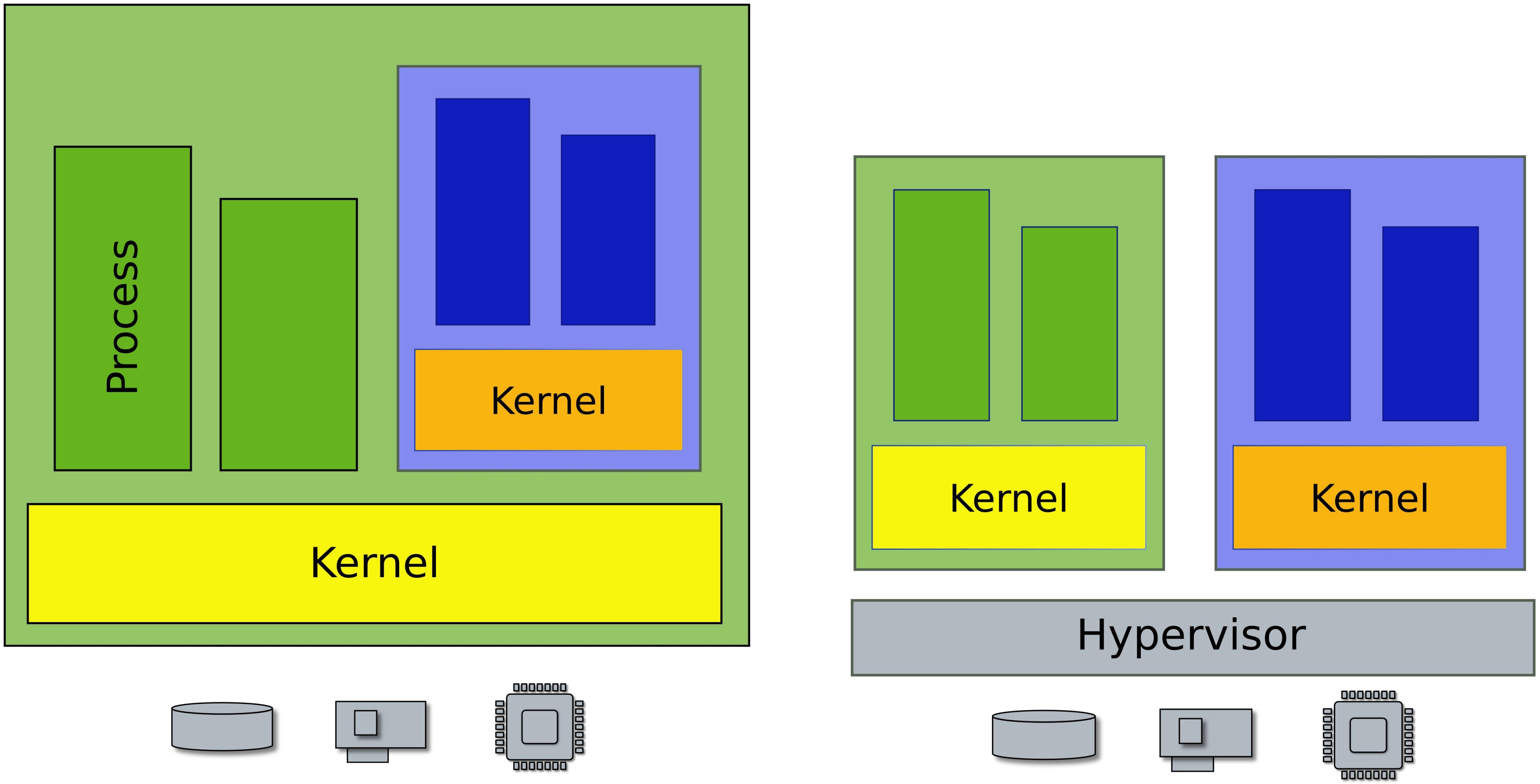
Anton Burtsev

March, 2013

Traditional operating system



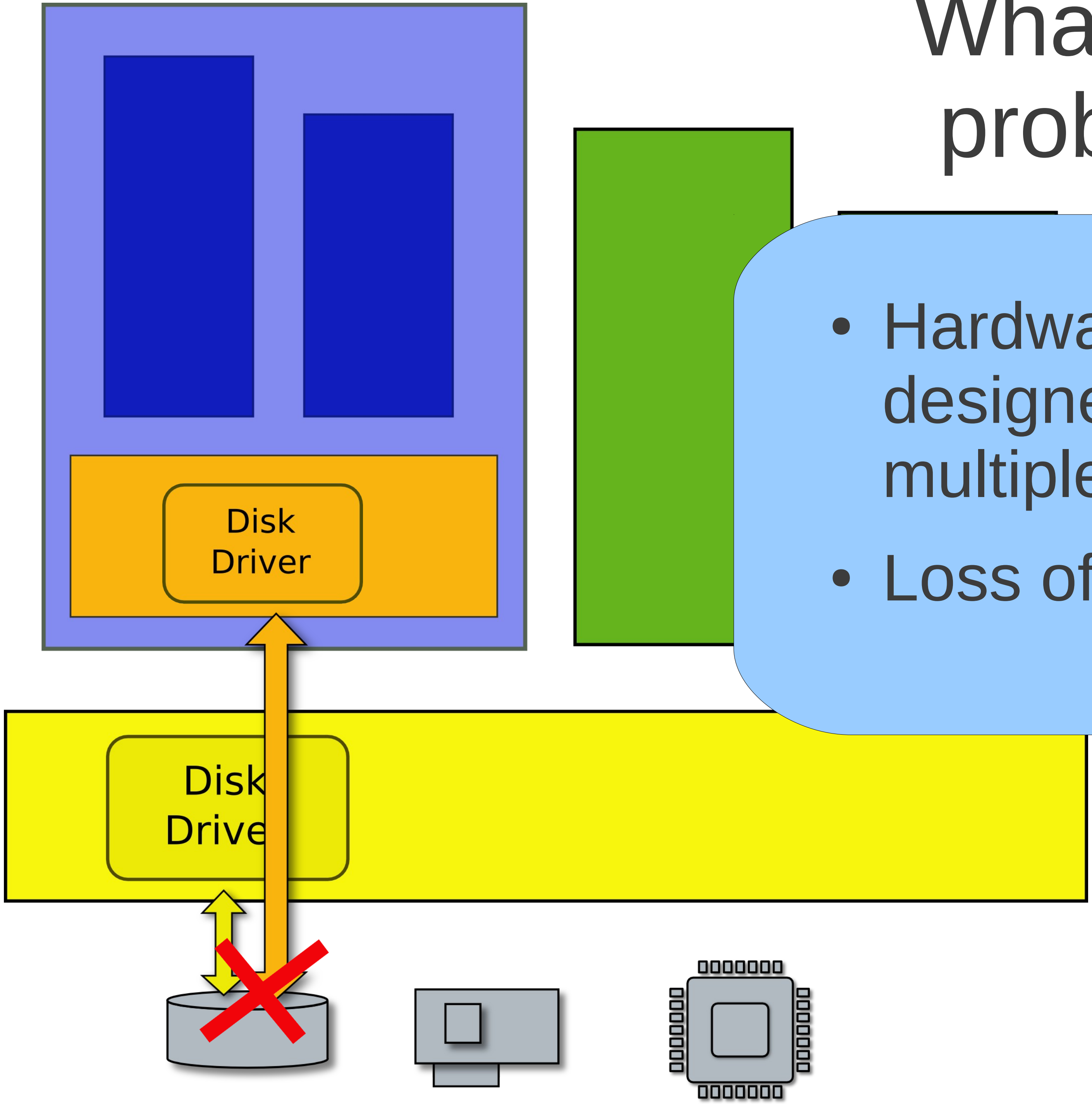
Virtual machines



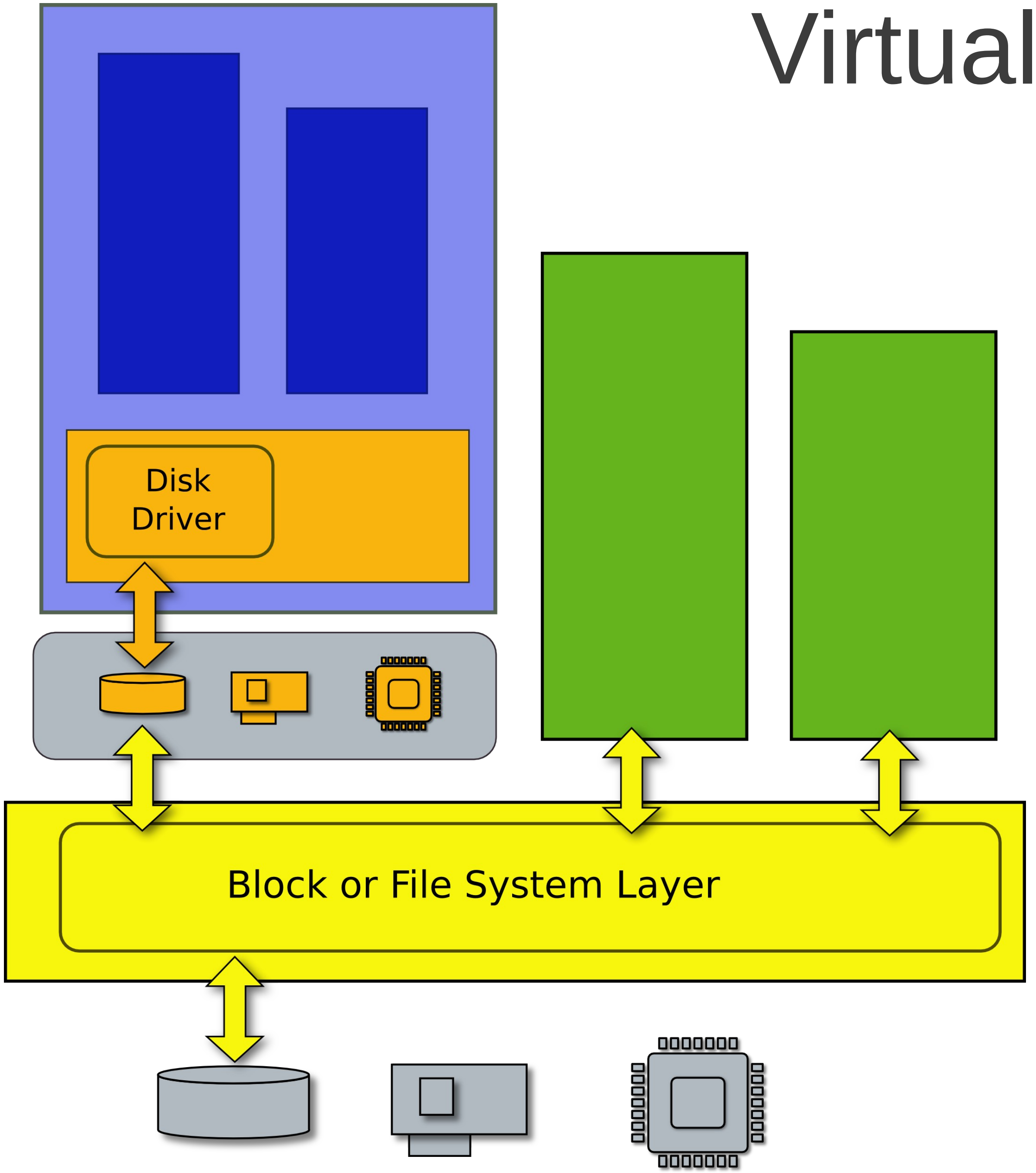
A bit of history

- Virtual machines were popular in 60s-70s
  - Share resources of mainframe computers [Goldberg 1974]
  - Run multiple single-user operating systems
- Interest is lost by 80s-90s
  - Development of multi-user OS
  - Rapid drop in hardware cost
- Hardware support for virtualizaiton is lost

What is the problem?



- Hardware is not designed to be multiplexed
- Loss of isolation

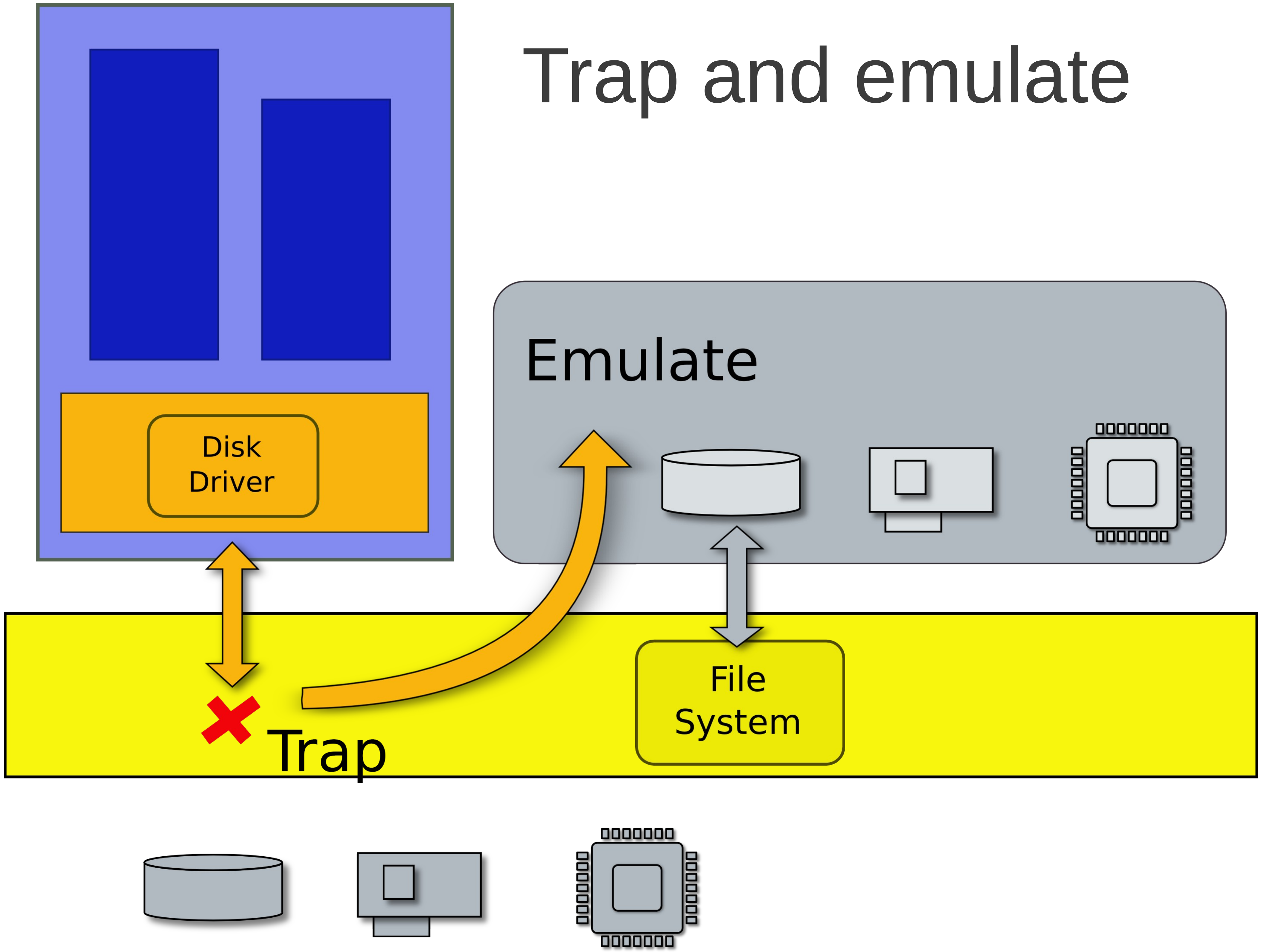


Virtual machine

Efficient duplicate of a real machine

- Compatibility
- Performance
- Isolation

Trap and emulate



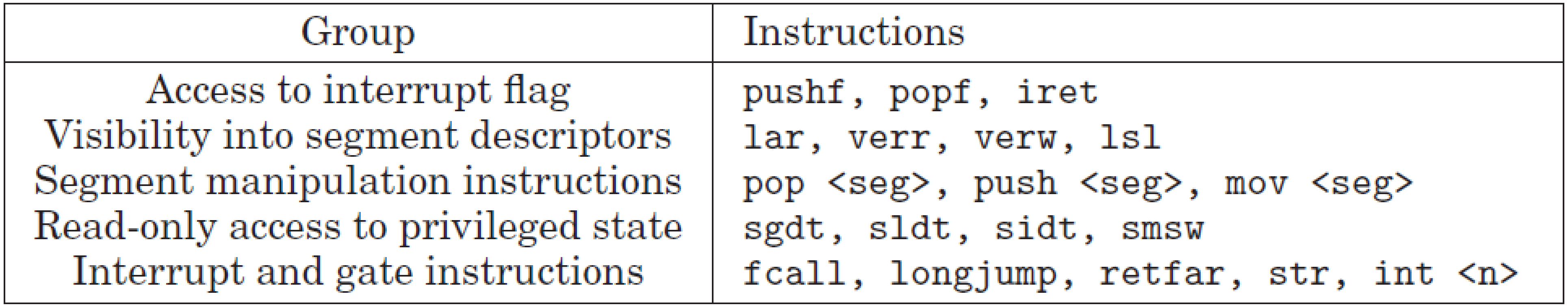
What needs to be emulated?

- CPU and memory
  - Register state
  - Memory state
- Memory management unit
  - Page tables, segments
- Platform
  - Interrupt controller, timer, buses
- BIOS
- Peripheral devices
  - Disk, network interface, serial line

x86 is not virtualizable

- Some instructions (*sensitive*) read or update the state of virtual machine and don't trap (*non-privileged*)
  - 17 sensitive, non-privileged instructions [Robin et al 2000]

x86 is not virtualizable (II)



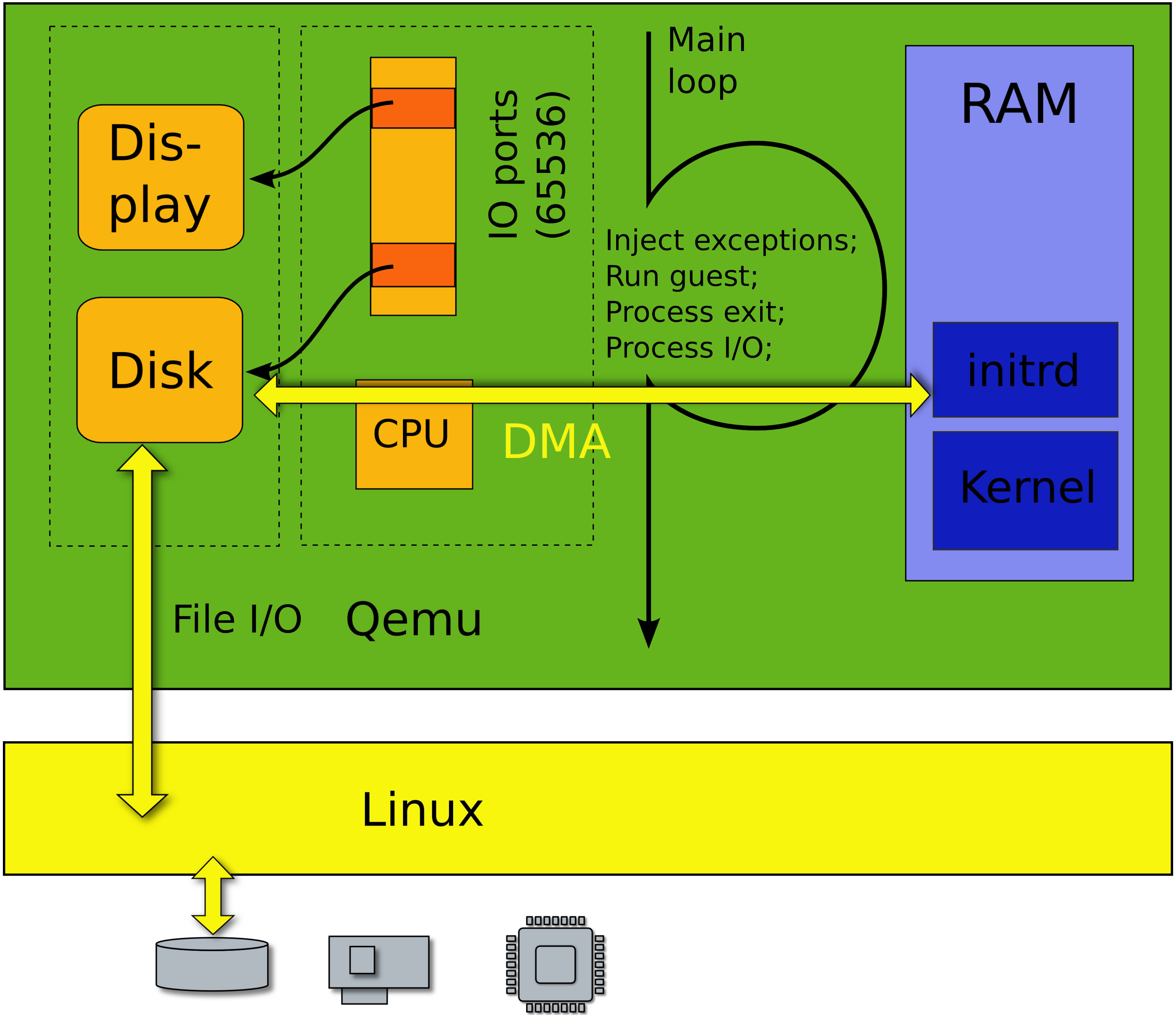
Group	Instructions
Access to interrupt flag	`pushf, popf, iret`
Visibility into segment descriptors	`lar, verr, verw, lsl`
Segment manipulation instructions	`pop <seg>, push <seg>, mov <seg>`
Read-only access to privileged state	`sgdt, sldt, sidt, smsw`
Interrupt and gate instructions	`fcall, longjump, retfar, str, int <n>`

- Examples
  - `popf` doesn't update interrupt flag (IF)
    - Impossible to detect when guest disables interrupts
  - `push %cs` can read code segment selector (%cs) and learn its CPL
    - Guest gets confused

Solution space

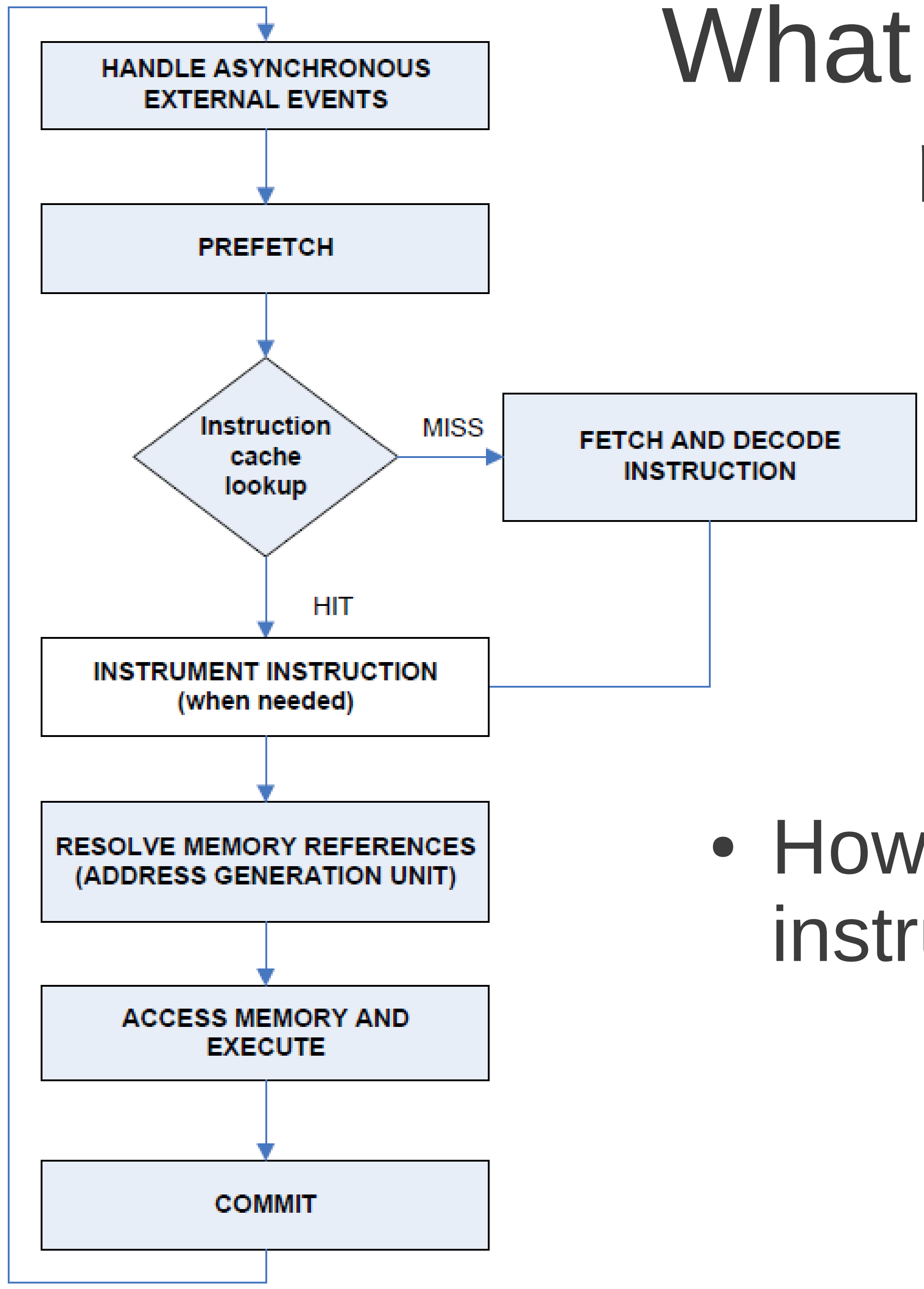
- Parse the instruction stream and detect all sensitive instructions dynamically
  - Interpretation (BOCHS, JSLinux)
  - Binary translation (VMWare, QEMU)
- Change the operating system
  - Paravirtualization (Xen, L4, Denali, Hyper-V)
- Make all sensitive instructions privileged!
  - Hardware supported virtualization (Xen, KVM, VMWare)
    - Intel VT-x, AMD SVM

Basic blocks of a virtual machine monitor: QEMU example


Main
loop
RAM
Dis-
play
IO ports
(65536)
Inject exceptions;
Run guest;
Process exit;
Process I/O;
Disk
initrd
CPU
DMA
Kernel
File I/O
Qemu
Linux

Interpreted execution: BOCHS, JSLinux

What does it mean to run guest?


HANDLE ASYNCHRONOUS EXTERNAL EVENTS
PREFETCH
Instruction cache lookup
MISS
FETCH AND DECODE INSTRUCTION
HIT
INSTRUMENT INSTRUCTION (when needed)
RESOLVE MEMORY REFERENCES (ADDRESS GENERATION UNIT)
ACCESS MEMORY AND EXECUTE
COMMIT

- Bochs internal emulation loop
- Similar to non-pipelined CPU like 8086

- How many cycles per instruction?

Binary translation: VMWare

```
int isPrime(int a) {
  for (int i = 2; i < a; i++) {
    if (a % i == 0) return 0;
  }
  return 1;
}
```

```
isPrime:   mov    %ecx, %edi ; %ecx = %edi (a)
           mov    %esi, $2   ; i = 2
           cmp    %esi, %ecx ; is i >= a?
           jge    prime      ; jump if yes
nexti:     mov    %eax, %ecx ; set %eax = a
           cdq               ; sign-extend
           idiv   %esi       ; a % i
           test   %edx, %edx ; is remainder zero?
           jz     notPrime   ; jump if yes
           inc    %esi       ; i++
           cmp    %esi, %ecx ; is i >= a?
           jl     nexti      ; jump if no
prime:     mov    %eax, $1   ; return value in %eax
           ret
notPrime:  xor    %eax, %eax ; %eax = 0
           ret
```

```
isPrime:   mov    %ecx, %edi ; %ecx = %edi (a)
           mov    %esi, $2   ; i = 2
           cmp    %esi, %ecx ; is i >= a?
           jge    prime      ; jump if yes
nexti:     mov    %eax, %ecx ; set %eax = a
           cdq               ; sign-extend
           idiv   %esi       ; a % i
           test   %edx, %edx ; is remainder zero?
           jz     notPrime   ; jump if yes
           inc    %esi       ; i++
           cmp    %esi, %ecx ; is i >= a?
           jl     nexti      ; jump if no
prime:     mov    %eax, $1   ; return value in %eax
           ret
notPrime:  xor    %eax, %eax ; %eax = 0
           ret
```

$$\Downarrow$$

```
isPrime':    mov %ecx, %edi     ; IDENT
             mov %esi, $2
             cmp %esi, %ecx
             jge [takenAddr]    ; JCC
             jmp [fallthrAddr]
```

```
isPrime':   *mov   %ecx, %edi    ; IDENT
             mov   %esi, $2
             cmp   %esi, %ecx
             jge   [takenAddr]   ; JCC
                                 ; fall-thru into next CCF
nexti':     *mov   %eax, %ecx    ; IDENT
             cdq
             idiv  %esi
             test  %edx, %edx
             jz    notPrime'     ; JCC
                                 ; fall-thru into next CCF
            *inc   %esi          ; IDENT
             cmp   %esi, %ecx
             jl    nexti'        ; JCC
             jmp   [fallthrAddr3]

notPrime': *xor   %eax, %eax    ; IDENT
             pop   %r11          ; RET
             mov   %gs:0xff39eb8(%rip), %rcx  ; spill %rcx
             movzx %ecx, %r11b
             jmp   %gs:0xfc7dde0(8*%rcx)
```

VMWare Workstation

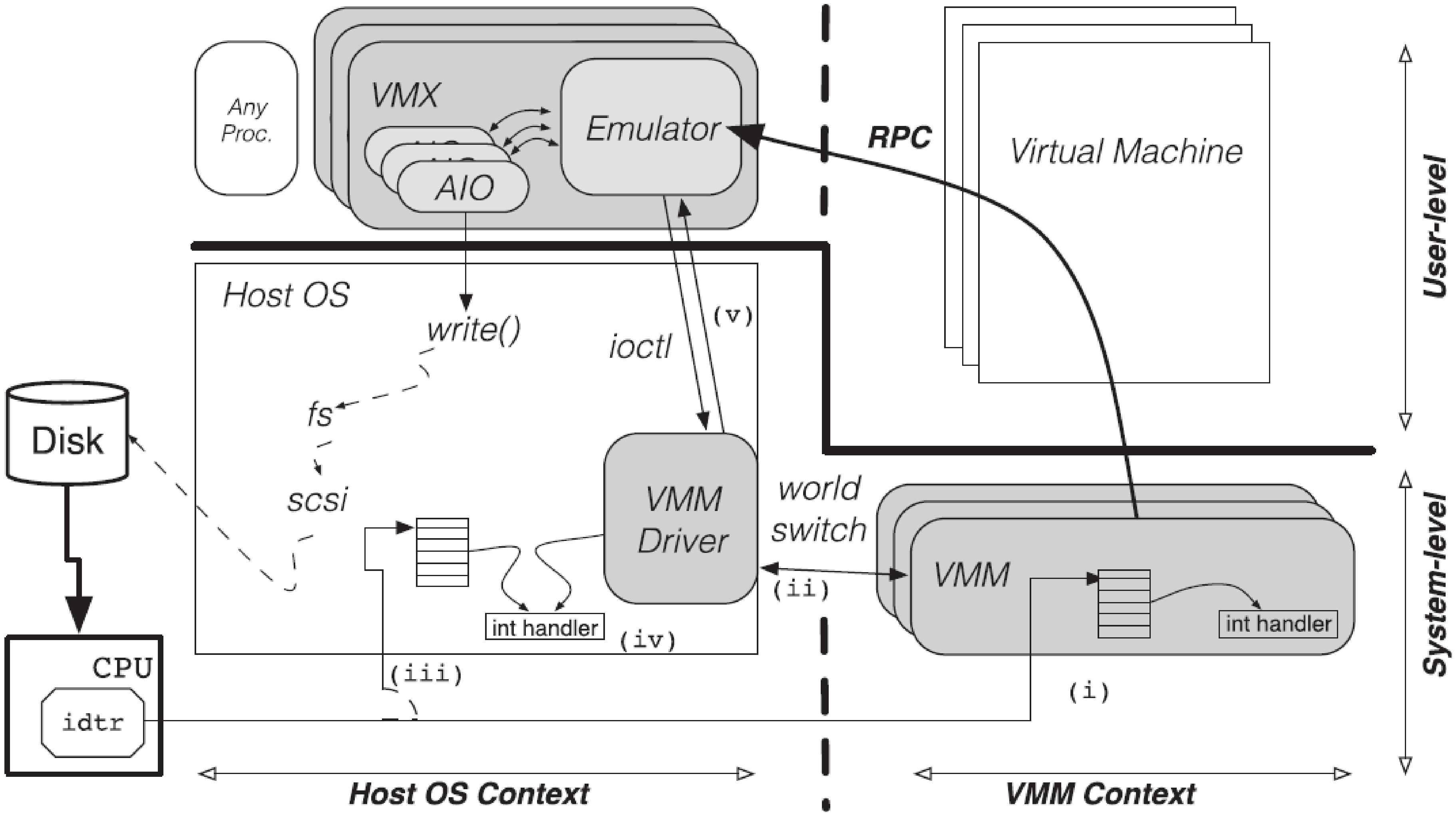
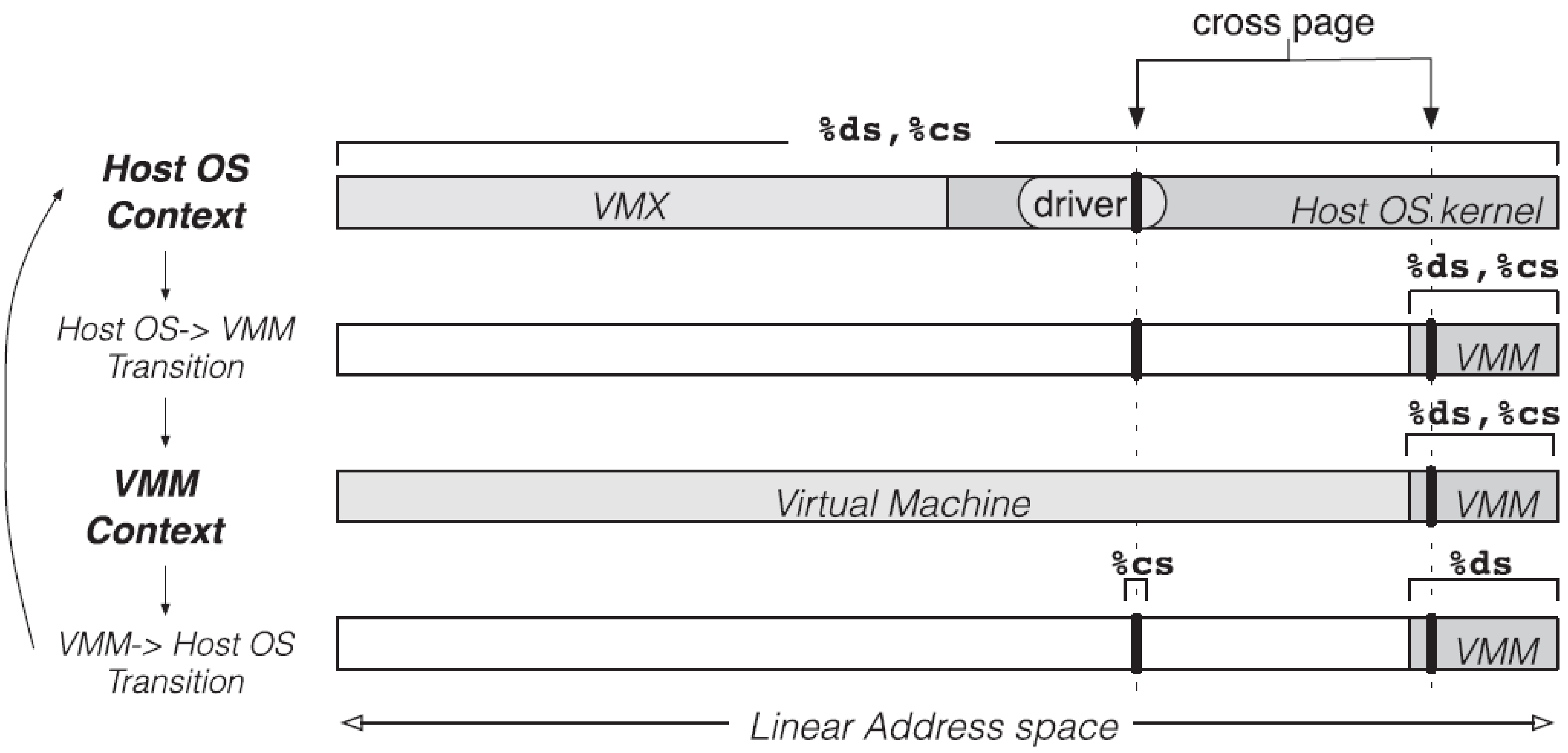


Fig. 2. The VMware Hosted Architecture. VMware Workstation consists of the three shaded components. The figure is split vertically between host operating system context and VMM context, and horizontally between system-level and user-level execution. The steps labeled (i)–(v) correspond to the execution that follows an external interrupt that occurs while the CPU is executing in VMM context.

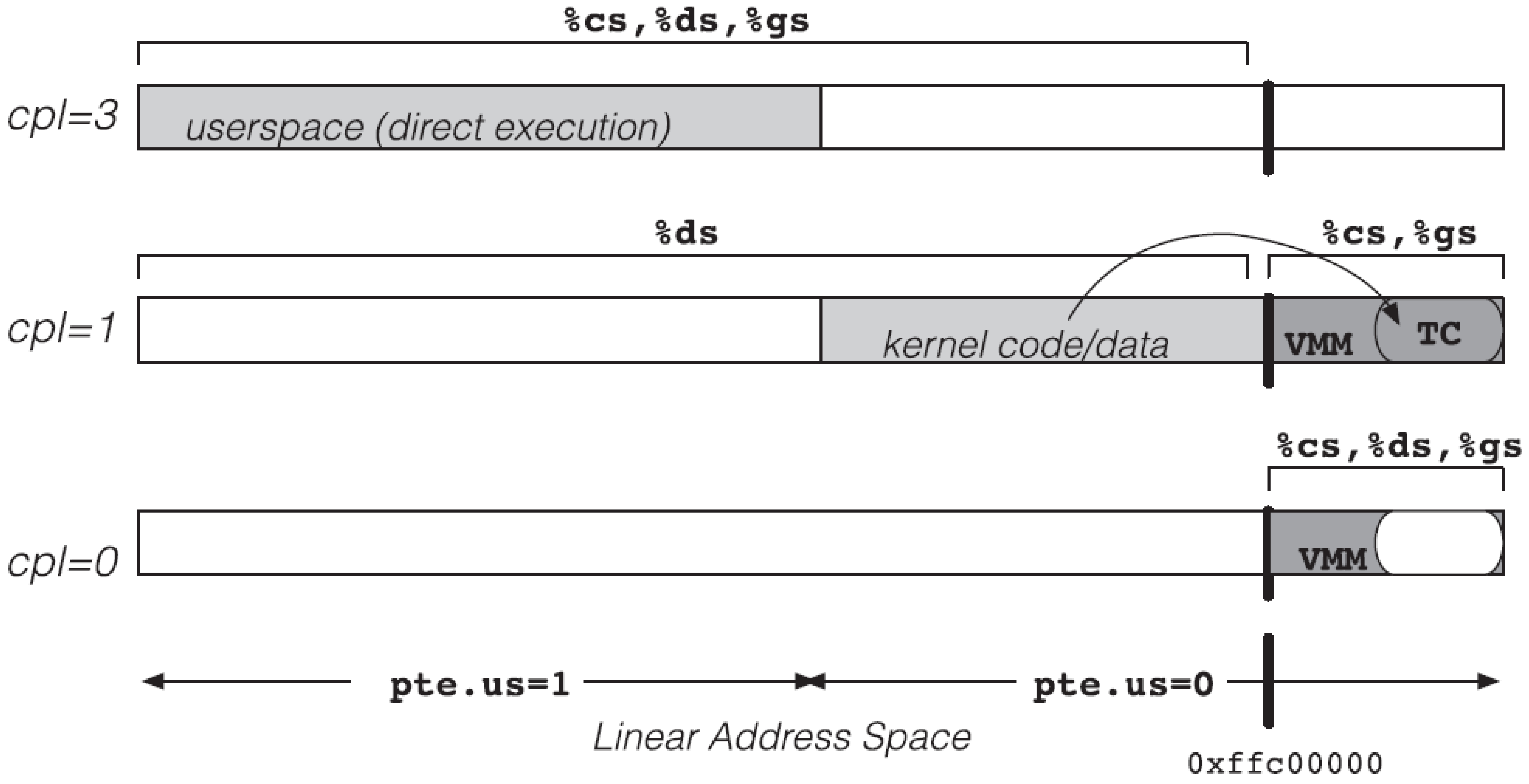
Address space during the world switch



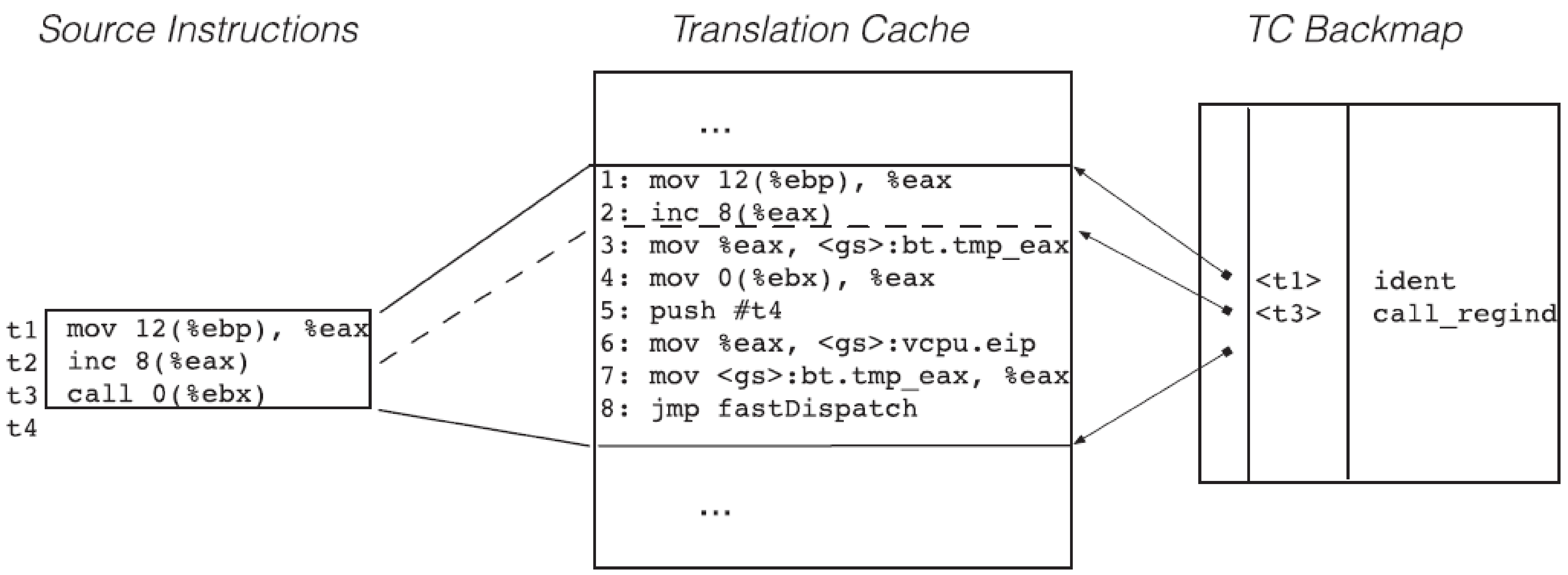
The world switch

- First, save the old processor state: general-purpose registers, privileged registers, and segment registers;
- Then, restore the new address space by assigning %cr3. All page table mappings immediately change, except the one of the cross page.
- Restore the global segment descriptor table register (%gdtr).
- With the %gdtr now pointing to the new descriptor table, restore %ds. From that point on, all data references to the cross page must use a different virtual address to access the same data structure. However, because %cs is unchanged, instruction addresses remain the same.
- Restore the other segment registers, %idtr, and the general-purpose registers.
- Finally, restore %cs and %eip through a longjump instruction.

Protecting the VMM



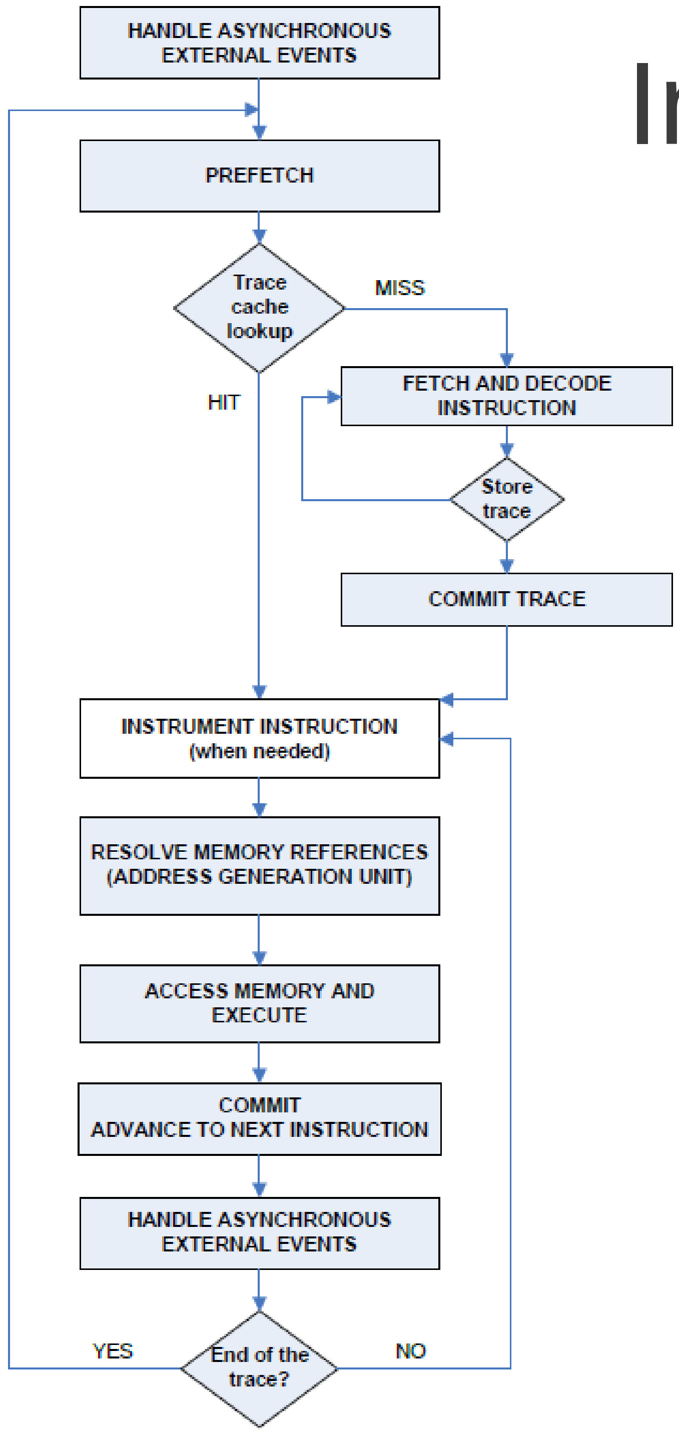
Translator continuations



```
t1  mov 12(%ebp), %eax
t2  inc 8(%eax)
t3  call 0(%ebx)
t4
```

```
1: mov 12(%ebp), %eax
2: inc 8(%eax)
3: mov %eax, <gs>:bt.tmp_eax
4: mov 0(%ebx), %eax
5: push #t4
6: mov %eax, <gs>:vcpu.eip
7: mov <gs>:bt.tmp_eax, %eax
8: jmp fastDispatch
```

Interpreted execution revisited: Bochs


HANDLE ASYNCHRONOUS EXTERNAL EVENTS
PREFETCH
Trace cache lookup
MISS
HIT
FETCH AND DECODE INSTRUCTION
Store trace
COMMIT TRACE
INSTRUMENT INSTRUCTION (when needed)
RESOLVE MEMORY REFERENCES (ADDRESS GENERATION UNIT)
ACCESS MEMORY AND EXECUTE
COMMIT ADVANCE TO NEXT INSTRUCTION
HANDLE ASYNCHRONOUS EXTERNAL EVENTS
YES
End of the trace?
NO

Instruction trace cache

- 50% of time in the main loop
  - Fetch, decode, dispatch
- Trace cache (Bochs v2.3.6)
  - Hardware idea (Pentium 4)
  - Trace of up to 16 instructions (32K entries)
- 20% speedup

Improve branch prediction

```
void BX_CPU_C::SUB_EdGd(bxInstruction_c *i)
{
  Bit32u op2_32, op1_32, diff_32;

  op2_32 = BX_READ_32BIT_REG(i->nnn());

  if (i->modC0()) {     // reg/reg format
    op1_32 = BX_READ_32BIT_REG(i->rm());
    diff_32 = op1_32 - op2_32;
    BX_WRITE_32BIT_REGZ(i->rm(), diff_32);
  }
  else {                // mem/reg format
    read_RMW_virtual_dword(i->seg(),
        RMAddr(i), &op1_32);
    diff_32 = op1_32 - op2_32;
    Write_RMW_virtual_dword(diff_32);
  }
  SET_LAZY_FLAGS_SUB32(op1_32, op2_32,
        diff_32);
}
```

- 20 cycles penalty on Core 2 Duo

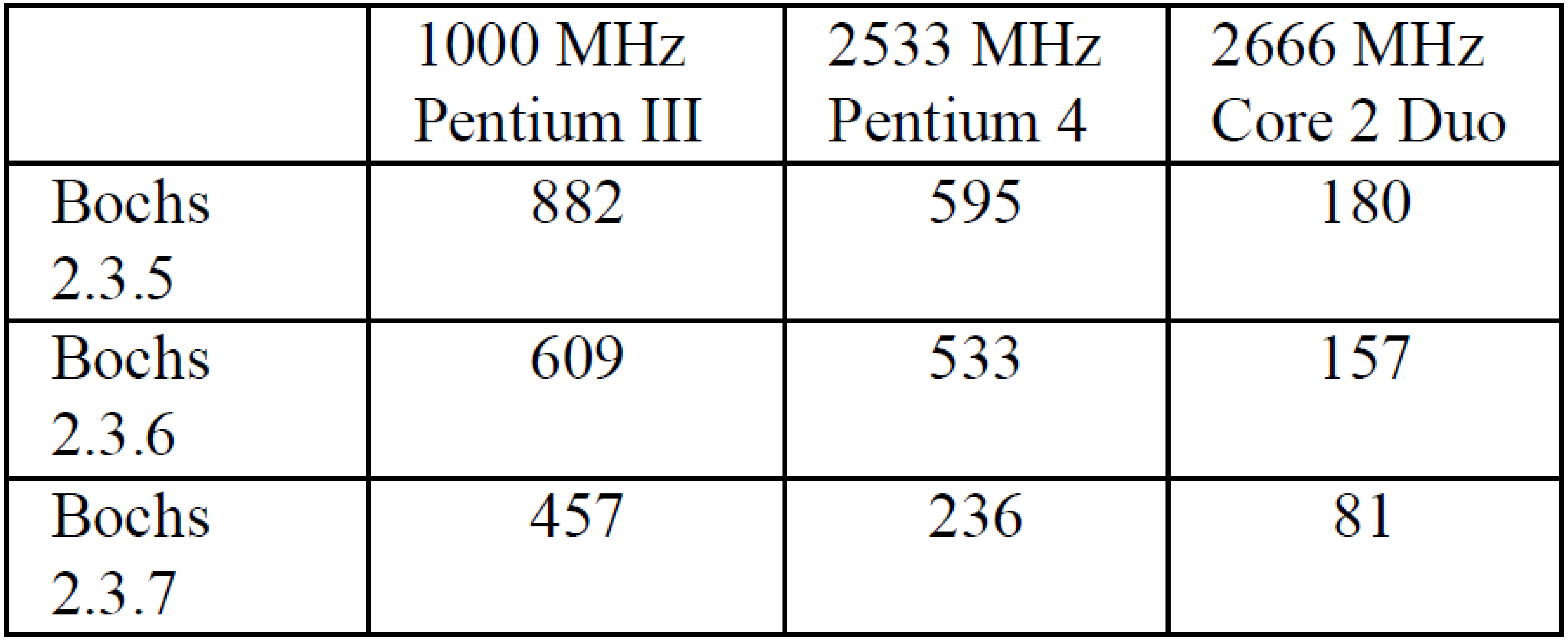
Improve branch prediction

- Split handlers to avoid conditional logic
  - Decide the handler at decode time (15% speedup)

Resolve memory references without misprediction

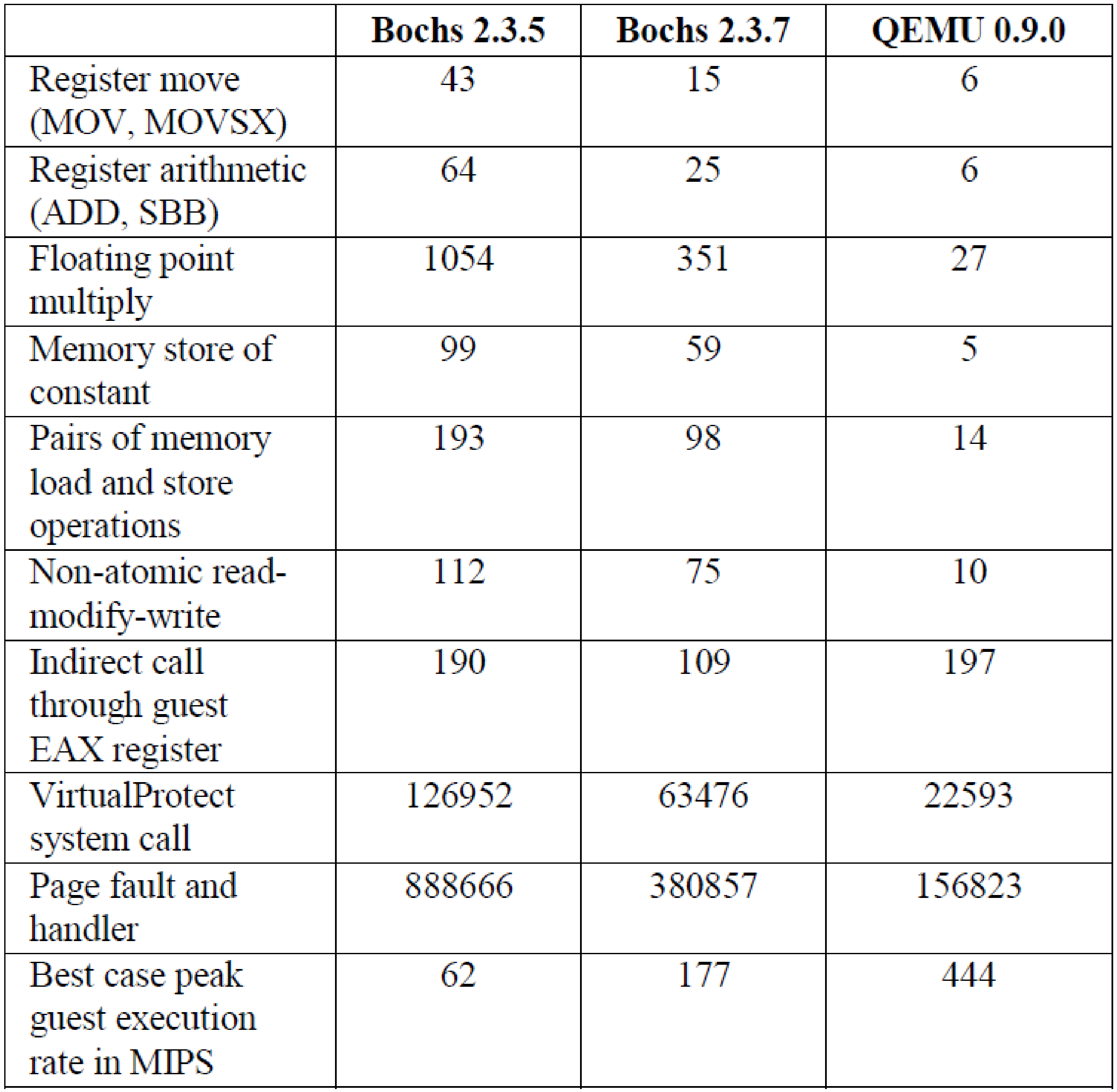
- Bochs v2.3.5 has 30 possible branch targets for the effective address computation
  - `Effective Addr = (Base + Index*Scale + Displacement) mod(2^AddrSize)`
  - e.g. `Effective Addr = Base`, `Effective Addr = Displacement`
  - 100% chance of misprediction
- Two techniques to improve prediction:
  - Reduce the number of targets: leave only 2 forms
  - Replicate indirect branch point
- 40% speedup

Time to boot Windows



	1000 MHz Pentium III	2533 MHz Pentium 4	2666 MHz Core 2 Duo
Bochs 2.3.5	882	595	180
Bochs 2.3.6	609	533	157
Bochs 2.3.7	457	236	81

Cycle costs



	Bochs 2.3.5	Bochs 2.3.7	QEMU 0.9.0
Register move (MOV, MOVSX)	43	15	6
Register arithmetic (ADD, SBB)	64	25	6
Floating point multiply	1054	351	27
Memory store of constant	99	59	5
Pairs of memory load and store operations	193	98	14
Non-atomic read-modify-write	112	75	10
Indirect call through guest EAX register	190	109	197
VirtualProtect system call	126952	63476	22593
Page fault and handler	888666	380857	156823
Best case peak guest execution rate in MIPS	62	177	444

References

- A Comparison of Software and Hardware Techniques for x86 Virtualization. Keith Adams, Ole Agesen, ASPLOS'06
- Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, Edward Y. Wang, ACM TCS'12.
- Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure. Darek Mihocka, Stanislav Shwartsman.