Computer Architecture

Multilevel paging, TLB and Caches

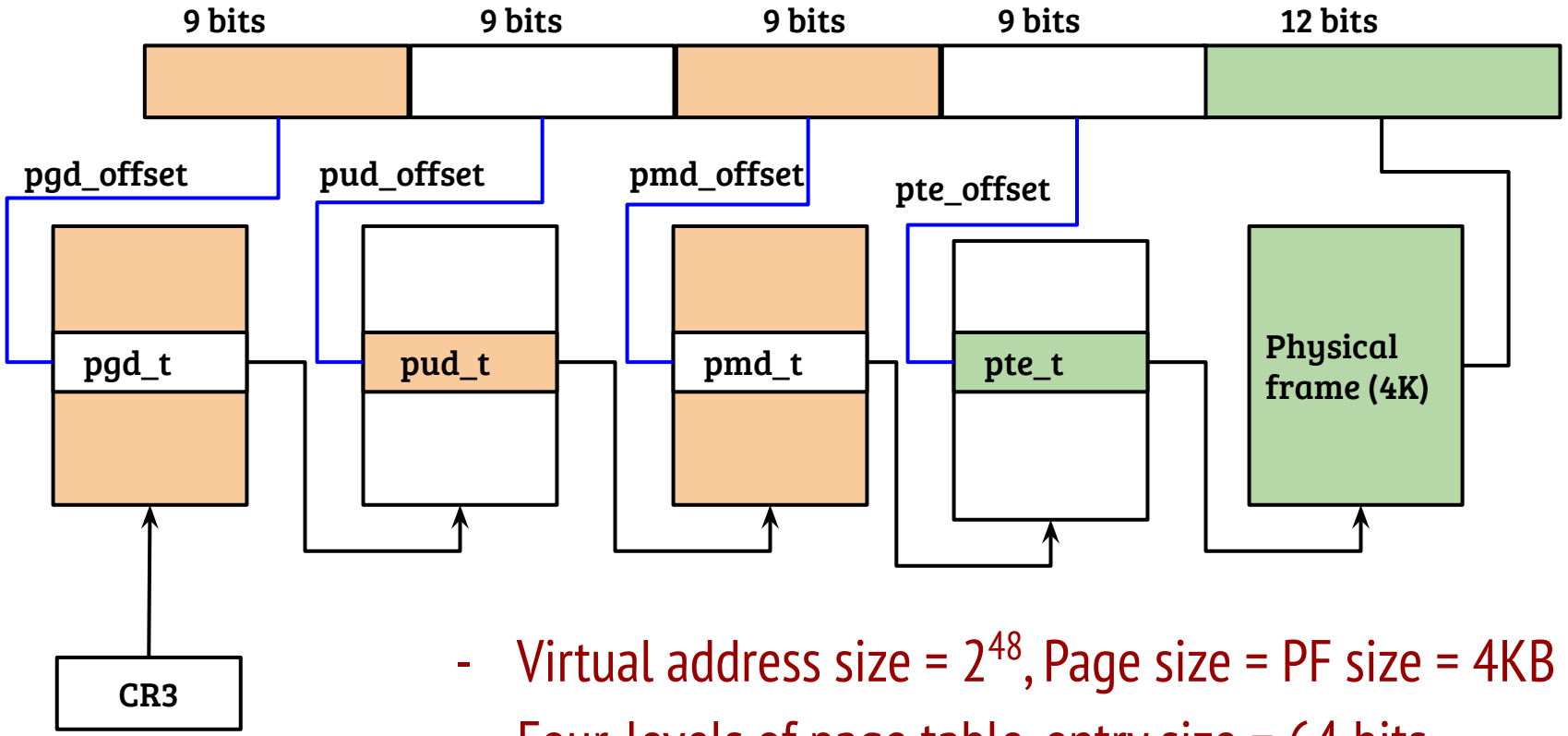
Debadatta Mishra, CSE, IITK

Paging

- The idea of paging
  - Partition the address space into fixed sized blocks (call it pages)
  - Physical memory partitioned in a similar way (call it page frames)
  - OS creates a mapping between *page* to *page frame* , H/W uses the mapping to translate VA to PA
- With increased address space size, single level page table entry is not feasible, because
  - Increasing page size (= frame size) increases internal fragmentation
  - Small pages may not be suitable to hold all mapping entries

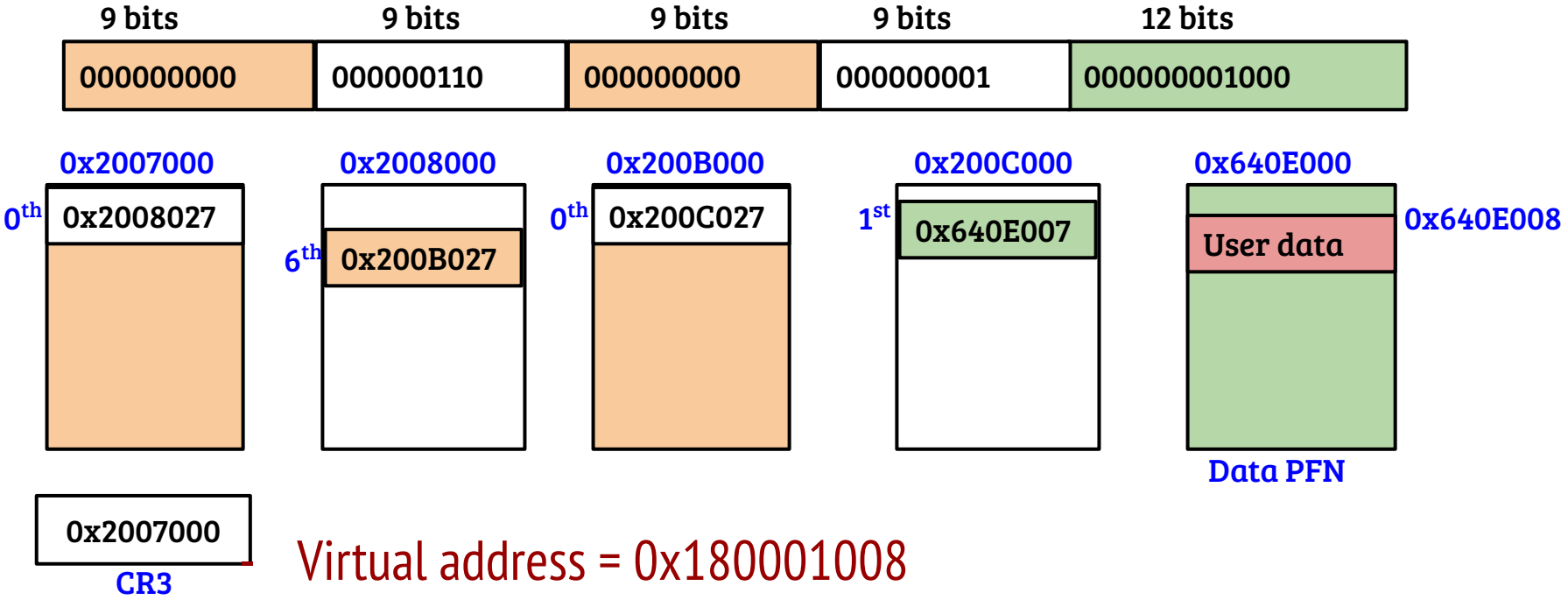
Agenda: Interplay of data caches and address translation

4-level page tables: 48-bit VA (Intel x86_64)



- Virtual address size = 2^{48}, Page size = PF size = 4KB
- Four-levels of page table, entry size = 64 bits

4-level page tables: example translation



Virtual address = 0x180001008

- Hardware translation by repeated access of page table stored in physical memory
- Page table entry: 12 bits LSB is used for access flags

Paging: translation efficiency

```
sum = 0;
for(ctr=0; ctr<10; ++ctr)
    sum += ctr;
```

```
0x20100:  mov $0, %rax;
0x20102:  mov %rax, (%rbp);     // sum=0
0x20104:  mov $0, %rcx;         // ctr=0
0x20106:  cmp $10, %rcx;        // ctr < 10
0x20109:  jge  0x2011f;         // jump if >=
0x2010f:  add %rcx, %rax;
0x20111:  mov %rax, (%rbp);     // sum += ctr
0x20113:  inc %rcx              // ++ctr
0x20115:  jmp 0x20106           // loop
0x2011f:  ..............
```

- Considering four-level page table, how many memory accesses are required (for translation) during the execution of the above code?

Paging: translation efficiency

```
0x20100:  mov $0, %rax;
0x20102:  mov %rax, (%rbp):     // sum=0
```

- Instruction execution: Loop = 10 * 6, Others = 2 + 3
  - Memory accesses during translation = 65 * 4 = 260
- Data/stack access: Initialization = 1, Loop = 10
  - Memory accesses during translation = 11 * 4 = 44
- A lot of memory accesses (> 300) for address translation
- How many distinct pages are translated? Assume stack address range 0x7FFF000 - 0x8000000
- Considering four-level page table, how many memory accesses are required (for translation) during the execution of the above code?

Paging: translation efficiency

0x20100: mov $0, %rax;

- Instruction execution: Loop = 10 * 6, Others = 2 + 3
  - Memory accesses during translation = 65 * 4 = 260
- Data/stack access: Initialization = 1, Loop = 10
  - Memory accesses during translation = 11 * 4 = 44
- A lot of memory accesses (> 300) for address translation
- How many distinct pages are translated? Assume stack address range 0x7FFF000 - 0x8000000
- One code page (0x20) and one stack page (0x7FFF). Caching these translations, will save a lot of memory accesses.

Paging with TLB: translation efficiency

TLB

Page	PTE
0x20	0x750
0x7FFF	0x890

- TLB is a hardware cache which stores *Page* to *PFN* mapping
- For code, after first miss for instruction fetch, all accesses hit the TLB
- Similarly, considering the stack virtual address range as 0x7FFF000 - 0x8000000, one entry in TLB avoids page table walk after first miss

Paging with TLB: translation efficiency

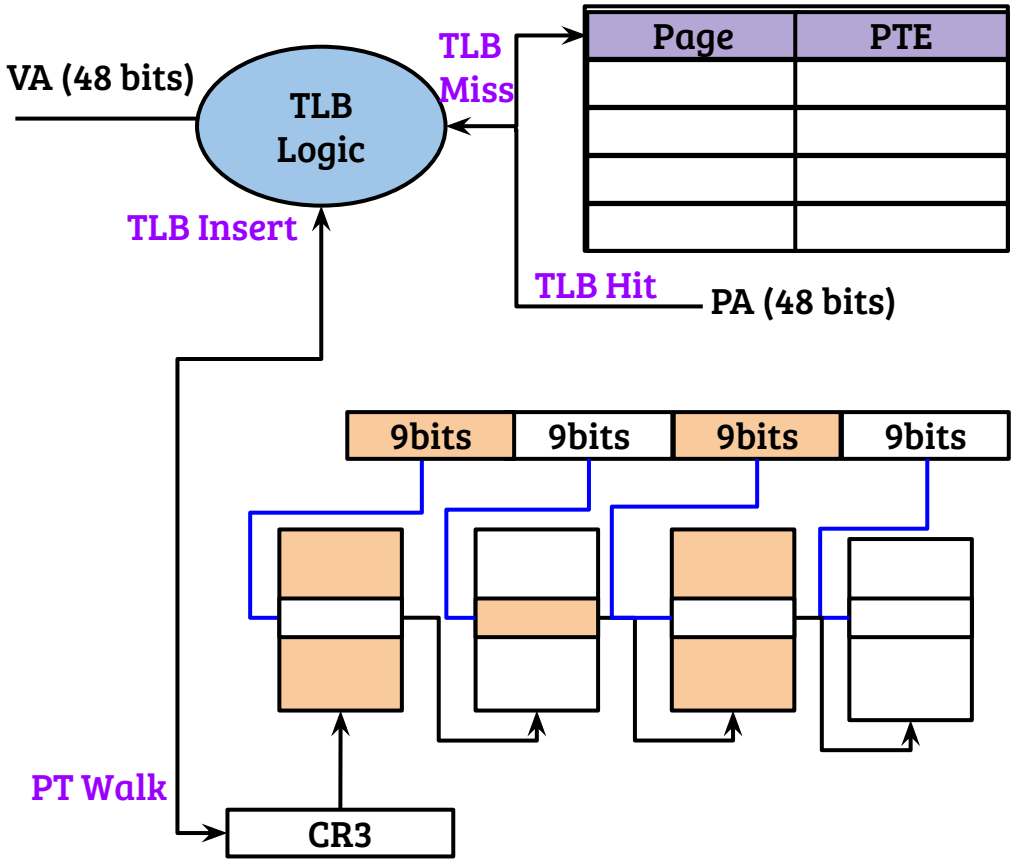
TLB

Page	PTE
0x20	0x750
0x7FFF	0x890

```
Translate(V){
    PageAddress P = V >> 12;
    TLBEntry entry = lookup(P);
    if (entry.valid) return entry.pte;
    entry = PageTableWalk(V);
    MakeEntry(entry);
    return entry.pte;
}
```

- TLB is a hardware cache which stores *Page* to *PFN* mapping
- After first miss for instruction fetch address, all others result in a TLB hit
- Similarly, considering the stack virtual address range as 0x7FFF000 - 0x8000000, one entry in TLB avoids page table walk after first miss

Address translation (TLB + PTW)



- TLB in the path of address translation
- Separate TLBs for instruction and data, multi-level TLBs
- In X86, OS can not make entries into the TLB directly, it can flush entries

Memory access (simplified serial view)

- Are there any differences in loading the data into L1 in two cases?
  - TLB Hit?
  - TLB Miss?

```
Addr Translate(V){
        PageAddress P = V >> 12;
        TLBEntry entry = lookup(P);
        if (entry.valid) return entry.pte;
        entry = PageTableWalk(V);
        MakeEntry(entry);
        return entry.pte;
}
Load(V){
   Addr PA = Translate( V);
   Load_L1(PA);
   …
}
```

Memory access (simplified serial view)

- Are there any differences in loading the data into L1 in two cases?
  - TLB Hit? (i) No access to page table entries required (ii) Accessed data is as recent as TLB entry
  - TLB Miss? (i) DRAM accesses for PTW (pte entries) (ii) Most likely the accessed data is not in the cache

```
Addr Translate(V){
        PageAddress P = V >> 12;
        TLBEntry entry = lookup(P);
        if (entry.valid) return entry.pte;
        entry = PageTableWalk(V);
        MakeEntry(entry);
        return entry.pte;
}
Load(V){
   Addr PA = Translate( V);
   Load_L1(PA);
   …
}
```

Translation Triggered Prefetching [1]

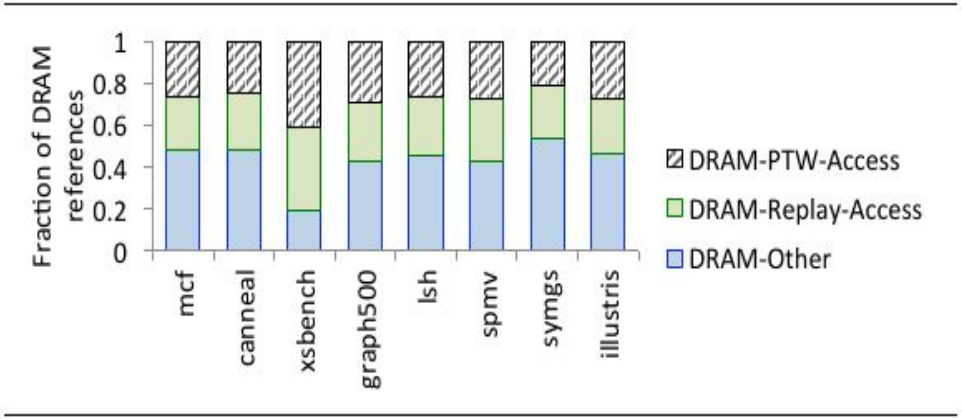


Figure 4. Fraction of total DRAM references devoted to page table walk accesses to DRAM (DRAM-PTW-access), replay accesses to DRAM (DRAM-Replay-Access), and other non-page table DRAM accesses (DRAM-Other).

- Replay accesses: DRAM accesses after a TLB miss followed by translation using PTW
- What is the expected replay access latency?

1. Translation-Triggered Prefetching, Abhishek Bhattacharjee, ASPLOS 2017

Translation Triggered Prefetching [1]

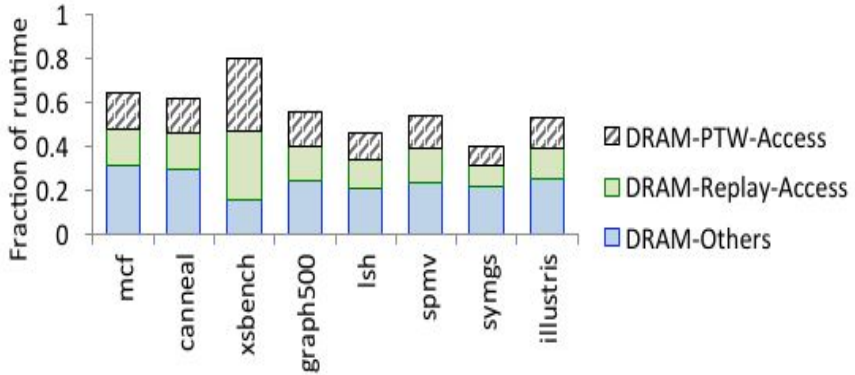
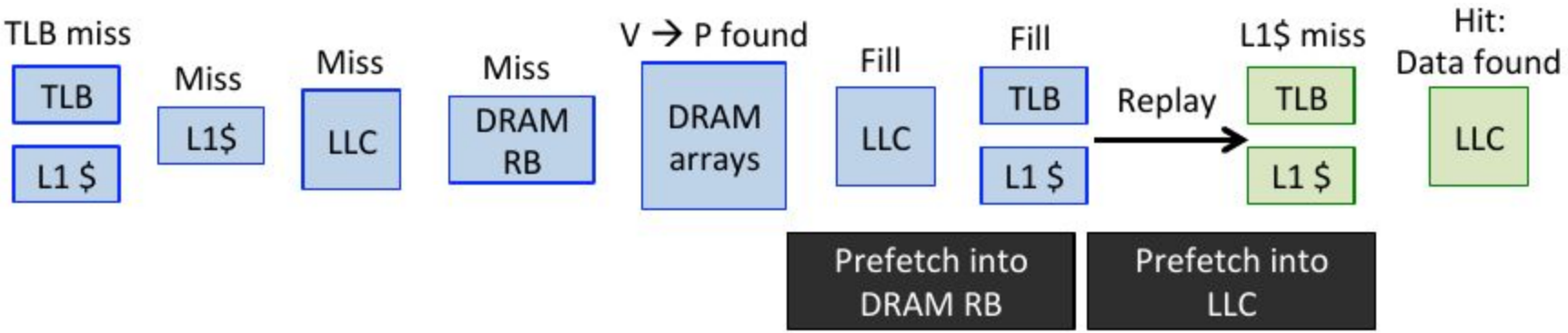


Figure 1. Fraction of total application runtime for page table accesses to DRAM (DRAM-PTW-Access), replayed accesses to DRAM (DRAM-Replay-Access), and other non-page table DRAM accesses (DRAM-Other).

- Replay accesses: DRAM accesses after a TLB miss followed by translation using PTW
- What is the expected replay access latency? If the translation is a miss, mostly the data access is a miss at all cache levels
- How to improve replay access latency?

1. Translation-Triggered Prefetching, Abhishek Bhattacharjee, ASPLOS 2017

Translation Triggered Prefetching [1]



- Main idea: Prefetch the data block into LLC as soon as the target physical address is known at the DRAM controller

1. Translation-Triggered Prefetching, Abhishek Bhattacharjee, ASPLOS 2017