TEXAS TECH UNIVERSITY
Department *of* Computer Science

CS5375 Computer Systems Organization and Architecture

Lecture 9

Instructor: Yong Chen, Ph.D.
Department of Computer Science
Texas Tech University
Yong.Chen@ttu.edu, 806-834-0284

Announcements

- Everyone's account is ready, you can try your account following the training videos

- Learn to use HPCC systems for both programming projects
  - Please watch Part 1 and Part 2 videos and check out slides too, to learn how to use the system
  - https://www.depts.ttu.edu/hpcc/about/training.php#new_user_training

  - Key parts you must study for now:
    - Part 1, "Logging and Using the Cluster"
    - Part 2, "Interactive Session"

  - If you're not familiar with Linux, then you must further study "Introduction to Linux"
  - https://www.depts.ttu.edu/hpcc/about/training.php#intro_linux

Review of Last Lecture

- Memory Technology and Optimizations
  - SRAM, DRAM, 3D-Stacked Memory (High Bandwidth Memory, HBM, Hybrid Memory Cube, HMC [1][2][3]), flash memory (and limitations), SSDs
- Optimizations of Cache Performance
  - Basic cache optimizations
  - Advanced optimizations
    - Reduce hit time (e.g., way prediction), increase bandwidth (non-blocking cache), reduce miss penalty (e.g., merging write buffers), reduce miss rate (compiler optimizations)

[1] HMC-Sim 3.0 Release, https://gc64.org/?p=303
[2] J. Leidel and Y. Chen. HMC-Sim-2.0: A Co-Design Infrastructure for Exploring Custom Memory Cube Operations. *The International Journal of Parallel Computing (ParCo)*, Volume: 68, Pages: 77 - 88, 2017.
[3] J. Leidel and Y. Chen. HMC-SIM: A Simulation Framework for Hybrid Memory Cube Devices. *Journal of Parallel Processing Letters*, Volume: 24, Issue: 04, Pages: 1465 - 1474, December 2014.

Outline

- Optimizations of Cache Performance (cont.)
- Virtual Memory and Virtual Machines

Reduce Miss Rate: Compiler Optimizations

- Loop Interchange
  - Swap nested loops to access memory in sequential order

```
for (j = 0; j < 100; j = j + 1)
    for (i = 0; i < 5000; i = i + 1)
        x[i][j] = 2 * x[i][j];
```

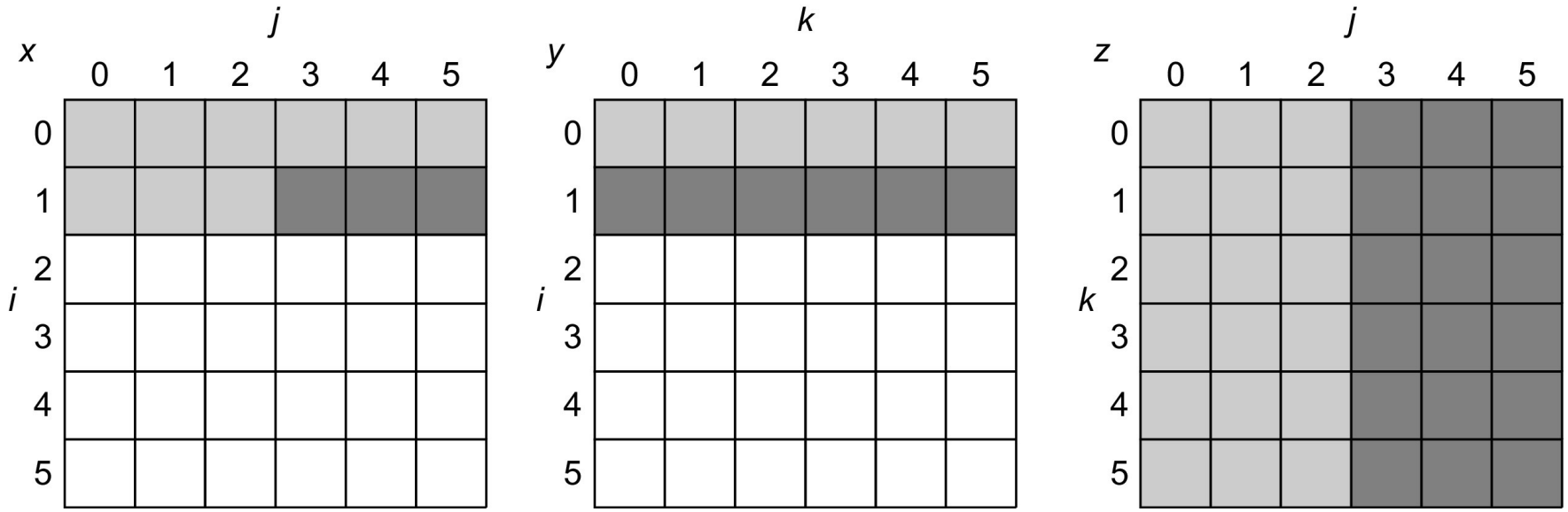
```
for (i = 0; i < 5000; i = i + 1)
    for (j = 0; j < 100; j = j + 1)
        x[i][j] = 2 * x[i][j];
```

The left-side code would skip through memory in strides of 100 words, while the right-side version accesses all the words in one cache block before going to the next block.

This optimization improves cache performance without affecting the number of instructions executed

Reduce Miss Rate: Compiler Optimizations (cont.)

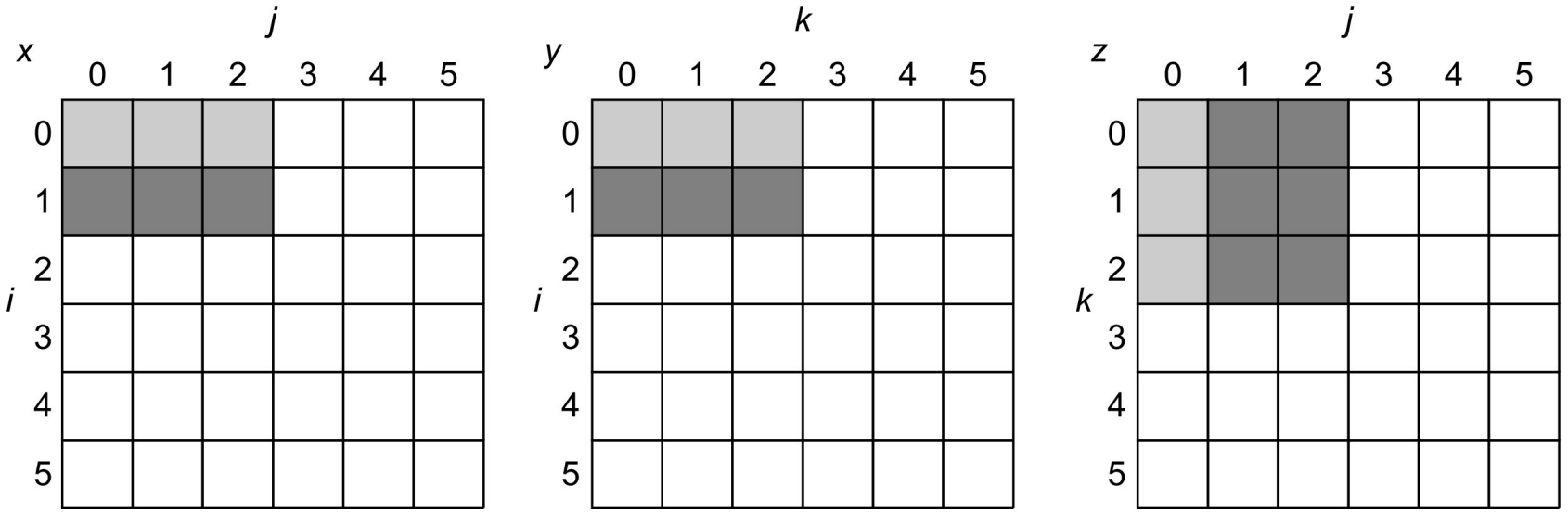
```
for (i = 0; i < N; i = i + 1)
  for (j = 0; j < N; j = j + 1)
  {
    r = 0;
    for (k = 0; k < N; k = k + 1)
      r = r + y[i][k]*z[k][j];
    x[i][j] = r;
};
```



Reduce Miss Rate: Compiler Optimizations (cont.)

```
for (jj = 0; jj < N; jj = jj + B)
  for (kk = 0; kk < N; kk = kk + B)
    for (i = 0; i < N; i = i + 1)
      for (j = jj; j < min(jj + B,N); j = j + 1)
      {
        r = 0;
        for (k = kk; k < min(kk + B,N); k = k + 1)
          r = r + y[i][k]*z[k][j];
        x[i][j] = x[i][j] + r;
};
```

- Blocking technique
- Instead of accessing entire rows or columns, subdivide matrices into blocks
- Improves locality of accesses



Summary

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined & banked caches	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs
HBM as additional level of cache		+/–	–	+	+	3	Depends on new packaging technology. Effects depend heavily on hit rate improvements

Virtual Memory

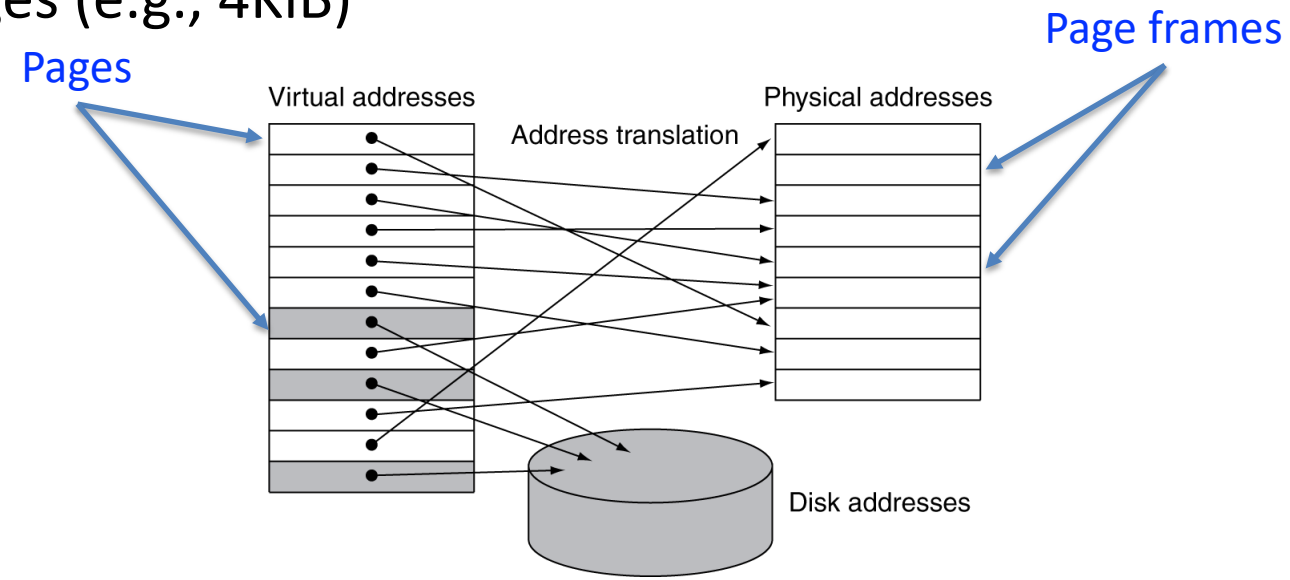
- Use main memory as a “cache” for secondary (disk) storage
  - Managed jointly by CPU hardware and the operating system (OS)

- Programs share main memory, but achieve protection via virtual memory
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs (keeps processes in their own memory space)

- CPU and OS translate virtual addresses to physical addresses
  - Virtual address: an address corresponds to a location in virtual space
  - Physical address: an address in main memory
  - VM “block” is called a page
  - VM translation “miss” is called a page fault

Address Translation (or address mapping)

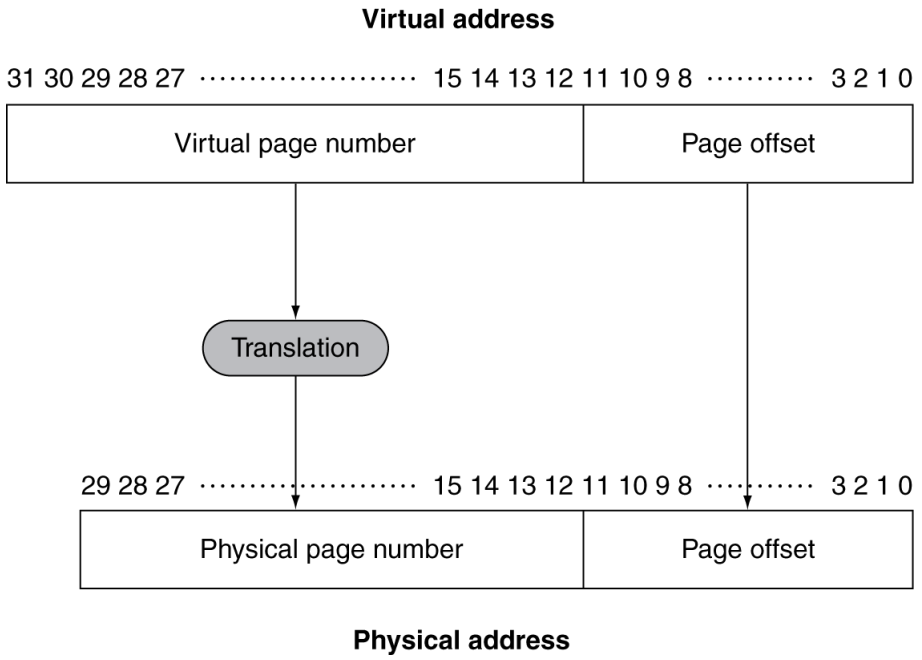
- Fixed-size pages (e.g., 4KiB)



CPU generates virtual addresses while the memory is accessed using physical addresses. Both the virtual memory and the physical memory are broken into same-size pages, so that a virtual page is mapped to a physical page.

Note that a virtual page can be absent from physical memory and resides on disk. On the other hand, physical pages can be shared by having two virtual address point to the same physical address to allow two processes to share data or code (e.g. DLLs).

Address Translation (or address mapping)



Mapping from a virtual to a physical address. The page size is 2^{12}=4 KiB. The number of physical pages allowed in memory is 2^{18}, since the physical page number has 18 bits in it. Thus, main memory can have at most 1 GiB, while the virtual address space is 4 GiB.

Page Fault Penalty

- On page fault, the page must be fetched from disk
  - Takes millions of clock cycles (disk access latency)
  - Handled by OS code

- Try to minimize page fault rate
  - Fully associative placement
  - Smart replacement algorithms

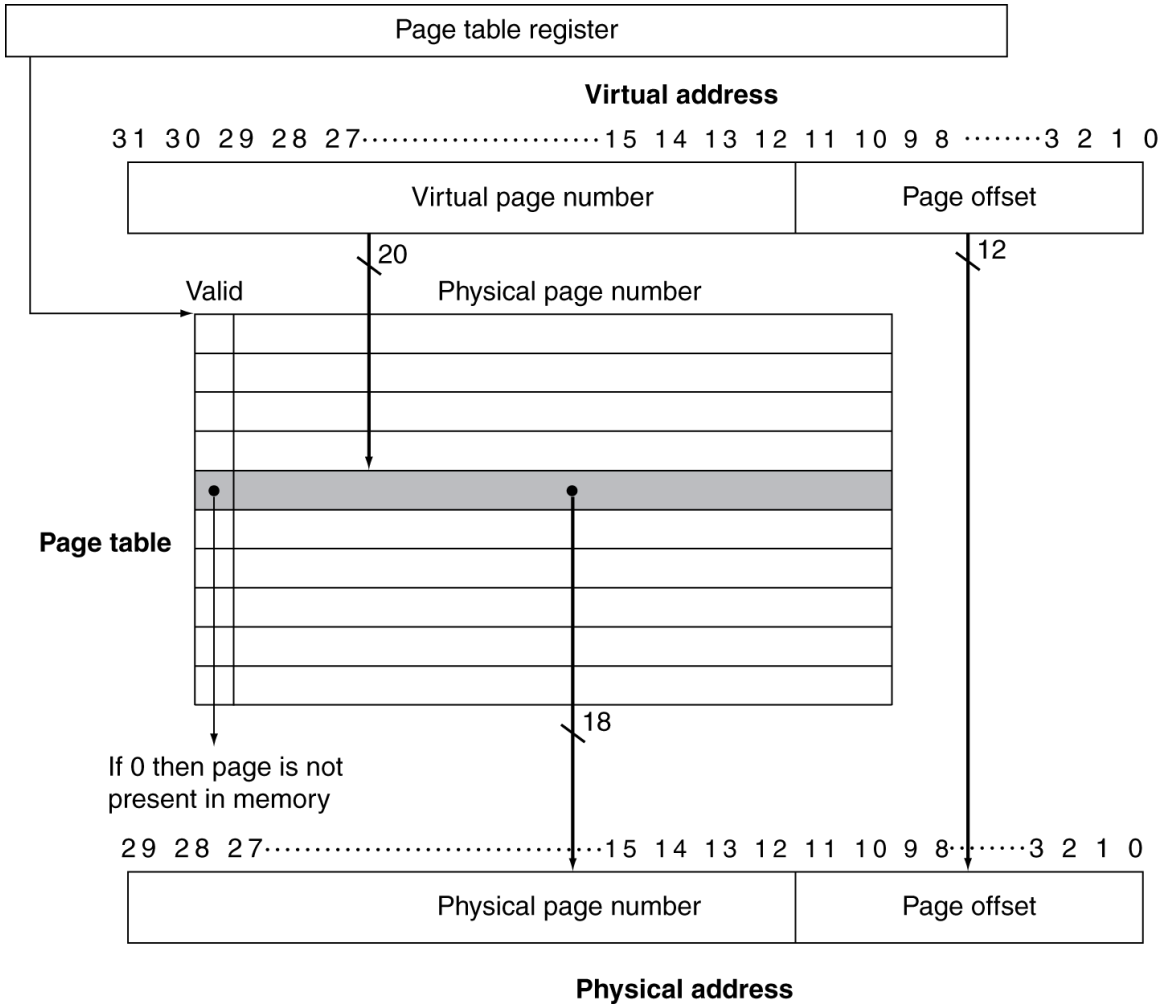
Page Tables

- Stores placement/mapping information
  - Array of page table entries, indexed by virtual page number
  - Page table register in CPU points to page table in physical memory

- If page is present in memory
  - PTE (page table entry) stores the physical page number
  - Plus, other status bits (referenced, dirty, …) for smart replacement algorithms

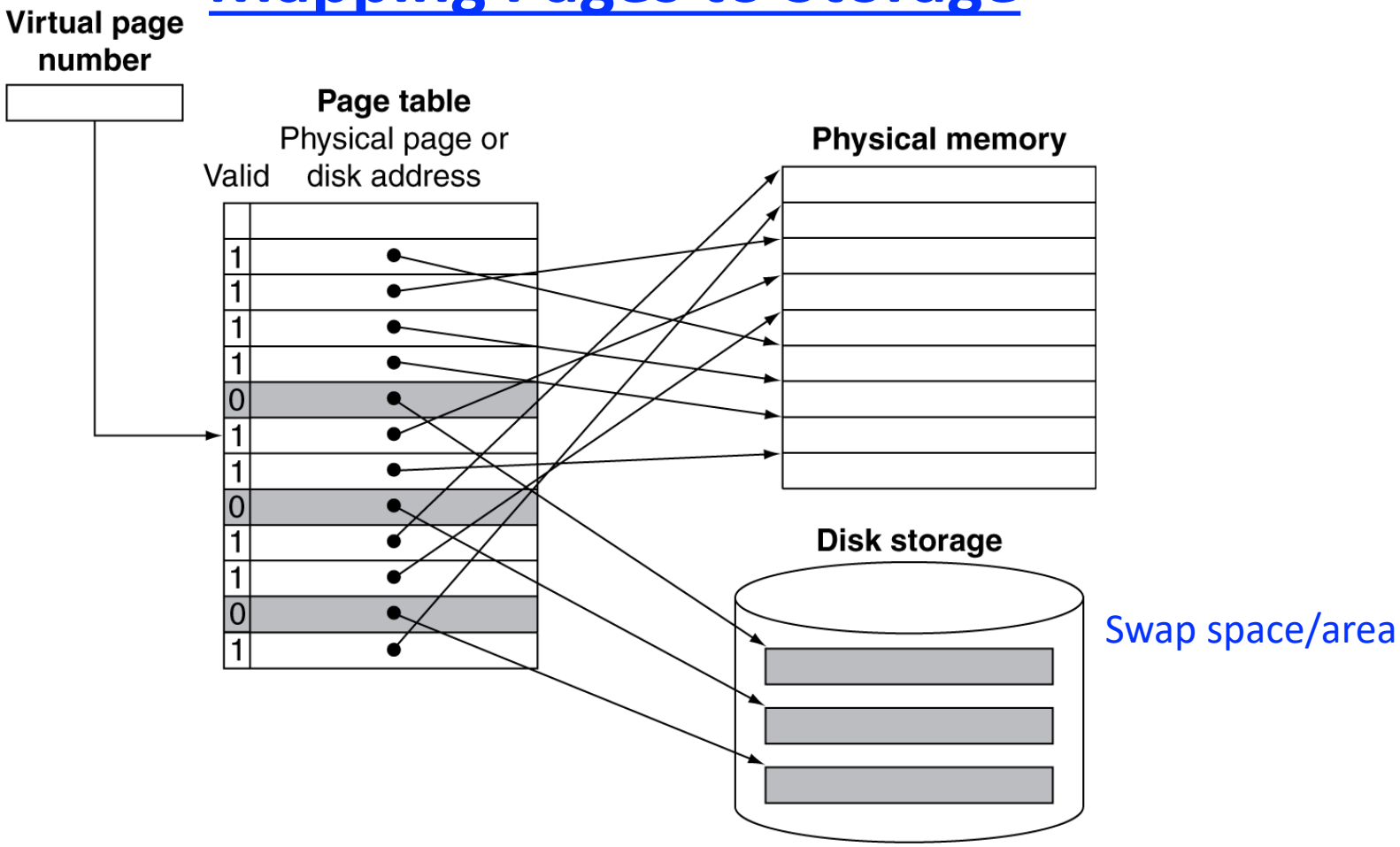
- If page is not present
  - PTE can refer to location in swap space on disk

Translation Using a Page Table



The number of entries in the page table is 2^{20}, or 1 million entries. The valid bit for each entry indicates whether the mapping is legal. If it is off , then the page is not present in memory.

Mapping Pages to Storage



The page table maps each page in virtual memory to either a page in main memory or a page stored on disk, which is the next level in the hierarchy.

Replacement and Writes

- To reduce page fault rate, prefer least-recently used (LRU) replacement, or an estimated LRU page
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 has not been used recently

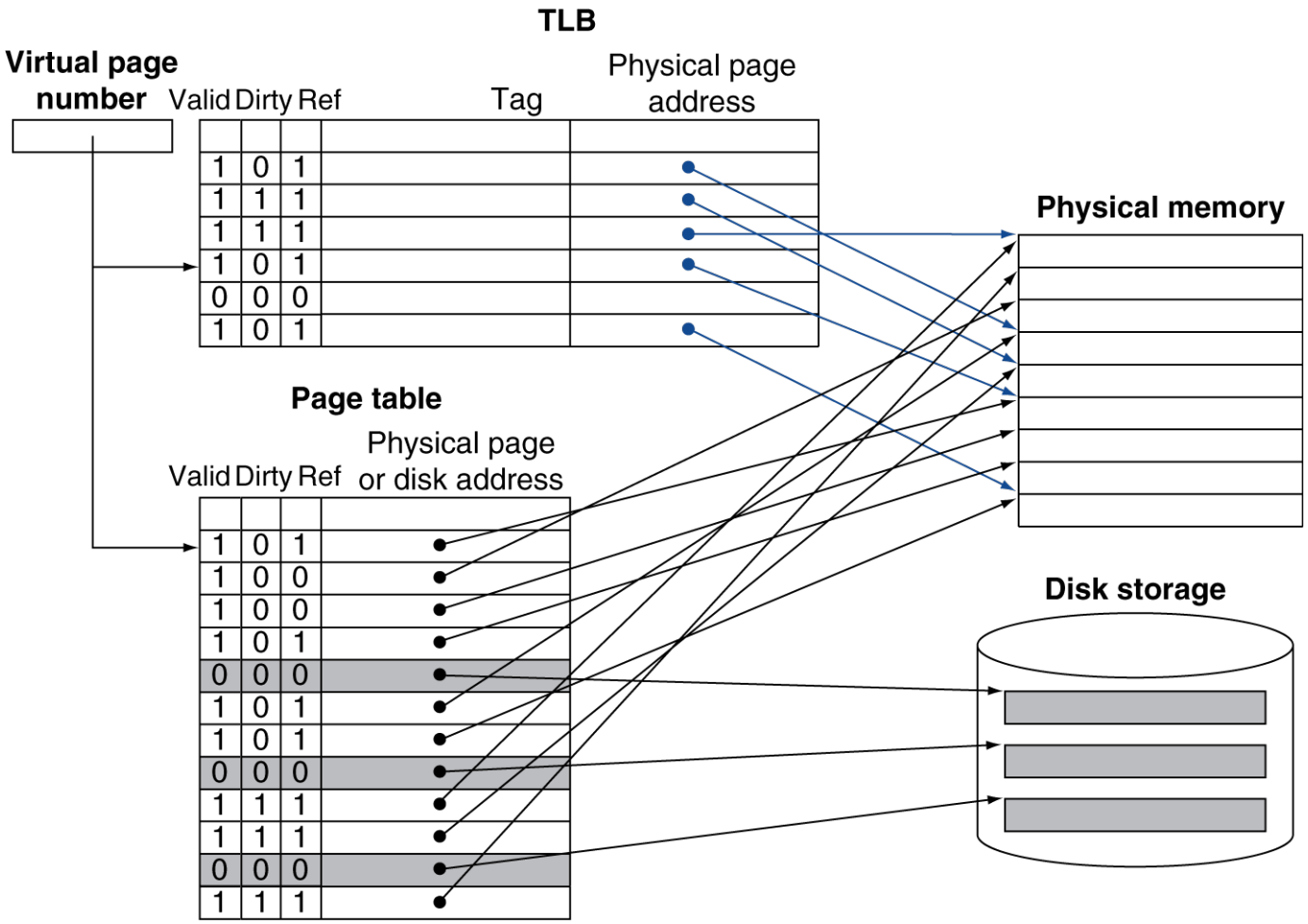
- Disk writes take millions of cycles
  - Write through is impractical, use write-back
  - Dirty bit in PTE set when page is written
  - If a page is replaced, and is dirty, then copying the page back to disk
    - Copying back an entire page is much more efficient than writing individual words back to the disk as in write-through case

Fast Translation Using a TLB

- Address translation would appear to require extra memory references
  - One memory reference is to access the PTE (because page table is in main memory)
  - Then the actual memory access (i.e., load/store) takes place

- But access to page tables has good locality
  - So, use a fast cache of PTEs within the CPU
  - Called a Translation Look-aside Buffer (TLB), a cache that keeps track recently used address mappings to avoid an access to the page table
  - Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10-100 cycles for miss (access the page table in main memory), 0.01%–1% miss rate

Fast Translation Using a TLB



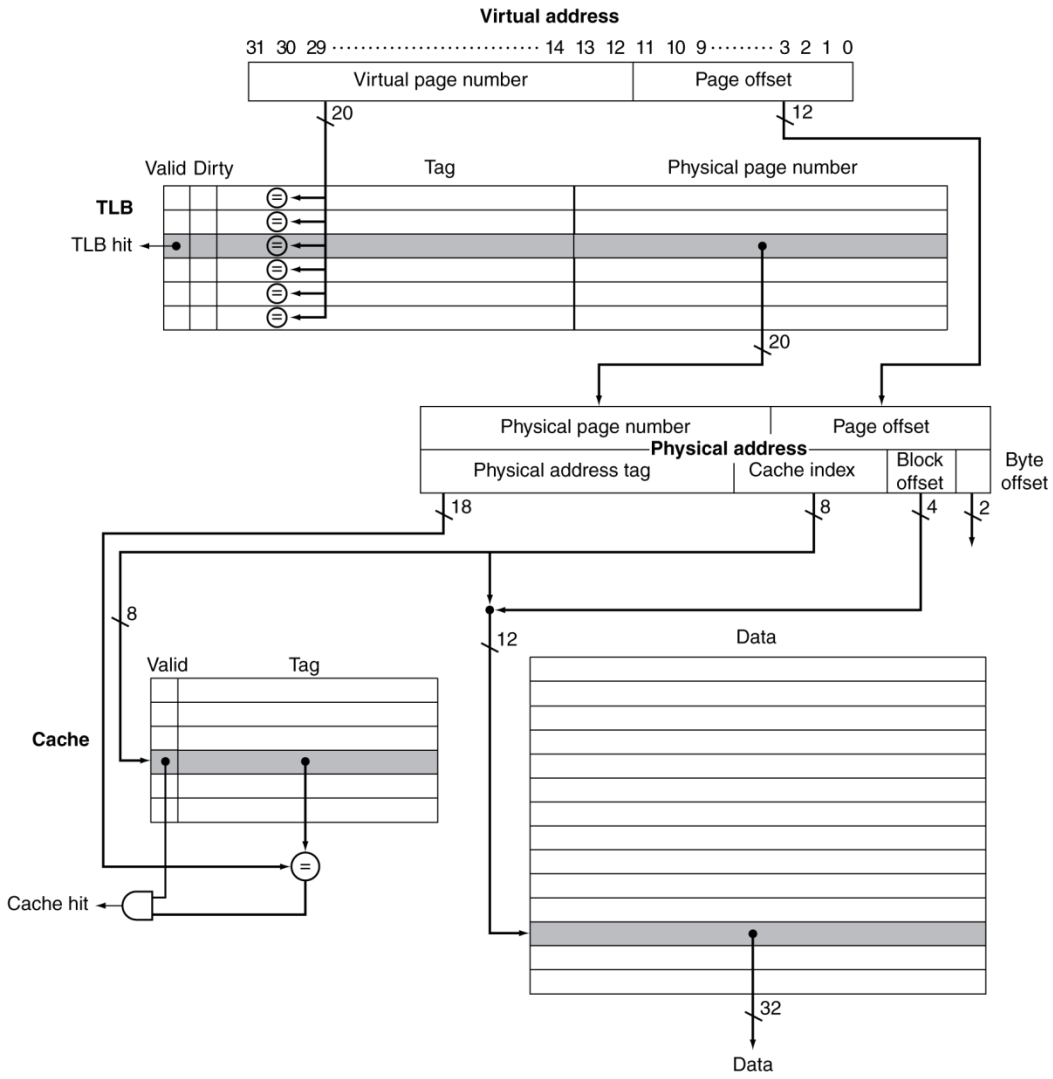
TLB Misses

- If page is in memory
  - Load the PTE from memory to TLB and retry
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler

- If page is not in memory (page fault)
  - OS uses faulting virtual address to find PTE
  - Locates the page on disk
  - Chooses a page to replace
    - If dirty, write to disk first
  - Reads the page into memory and updates page table
  - Make process runnable again
  - Restart from faulting instruction

TLB and Cache Interaction

- If cache tag uses physical address
  - Need to translate before cache lookup
- Alternative: use virtual address tag
  - Complications due to aliasing
    - Different virtual addresses for shared physical address
- TLB can be fully/set associative



Virtual Machines

- Supports isolation and security
- Sharing a computer among many unrelated users
- Enabled by raw speed of processors, making the overhead more acceptable
- Allows different ISAs and operating systems to be presented to user programs
  - "System Virtual Machines"
  - SVM software is called "virtual machine monitor" or "hypervisor"
  - Individual virtual machines run under the monitor are called "guest VMs"

Demo

- Log in the system following the instruction in Part 1, using your eRaider ID and password
  - E.g., `$ ssh yonchen@login.hpcc.ttu.edu`
- Request an interactive session for you to code, debug, test, etc., instead of overloading the head/login node
  - E.g., `$ interactive -p nocona`
- Check out sample code, compile, and run
  - E.g., `$ git clone https://github.com/githubyongchen/CS5375.git`

Readings

- Chapter 2, 2.3 – 2.6