Introduction to Operating Systems

CPSC/ECE 3220 Summer 2018

Lecture Notes
OSPP Chapter 9

(adapted by Mark Smotherman from Tom Anderson’s slides on OSPP web site)

Definitions

- Cache
  - General concept in memory hierarchy, not just a processor cache
  - Copy of data that is faster to access than the original
  - Hit: if cache has copy
  - Miss: if cache does not have copy
- Cache block
  - Unit of cache storage (multiple memory locations)
- Temporal locality
  - Programs tend to reference the same memory locations multiple times
  - Example: instructions in a loop
- Spatial locality
  - Programs tend to reference nearby locations
  - Example: data in a loop

Locality of Reference

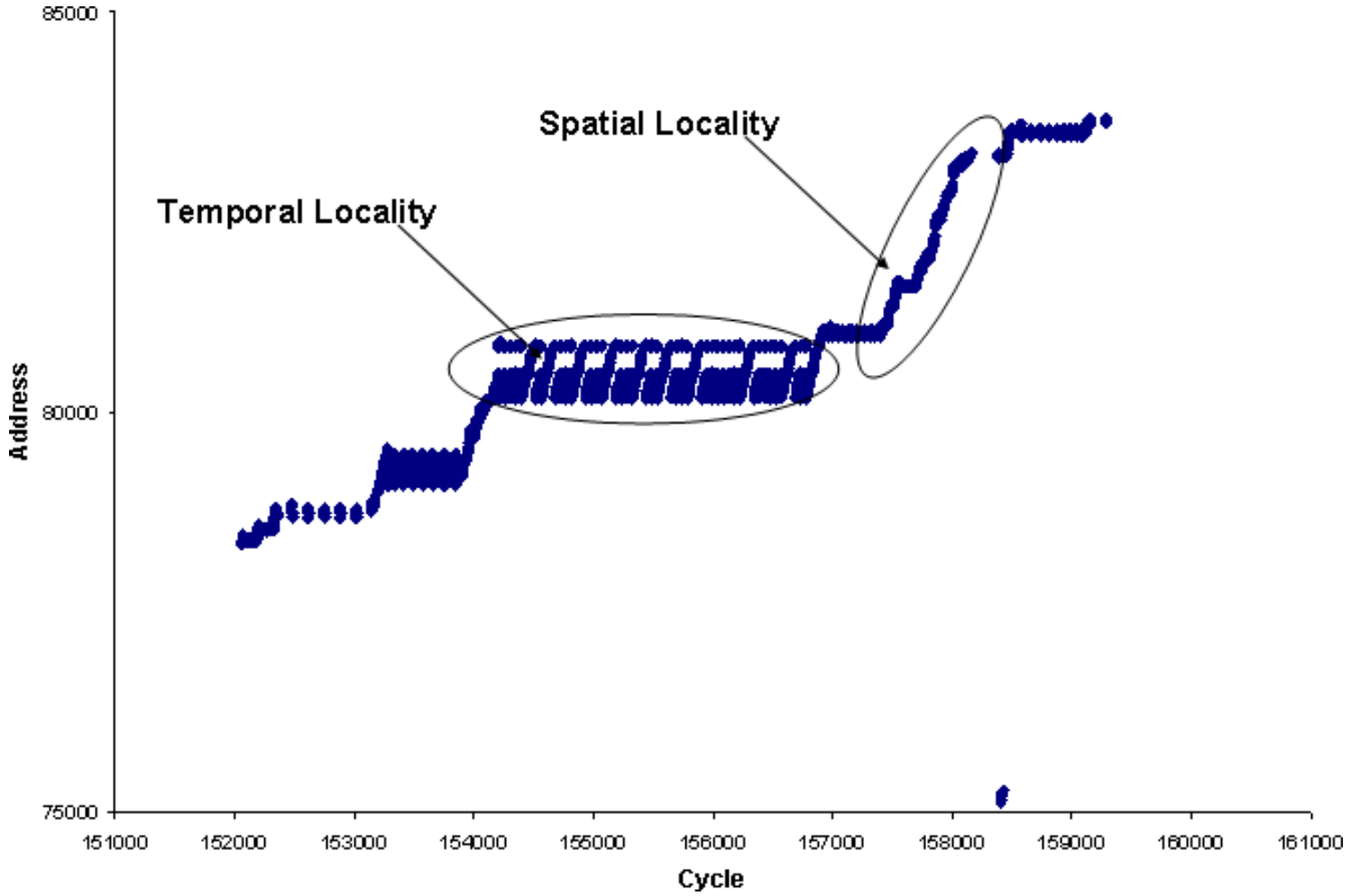
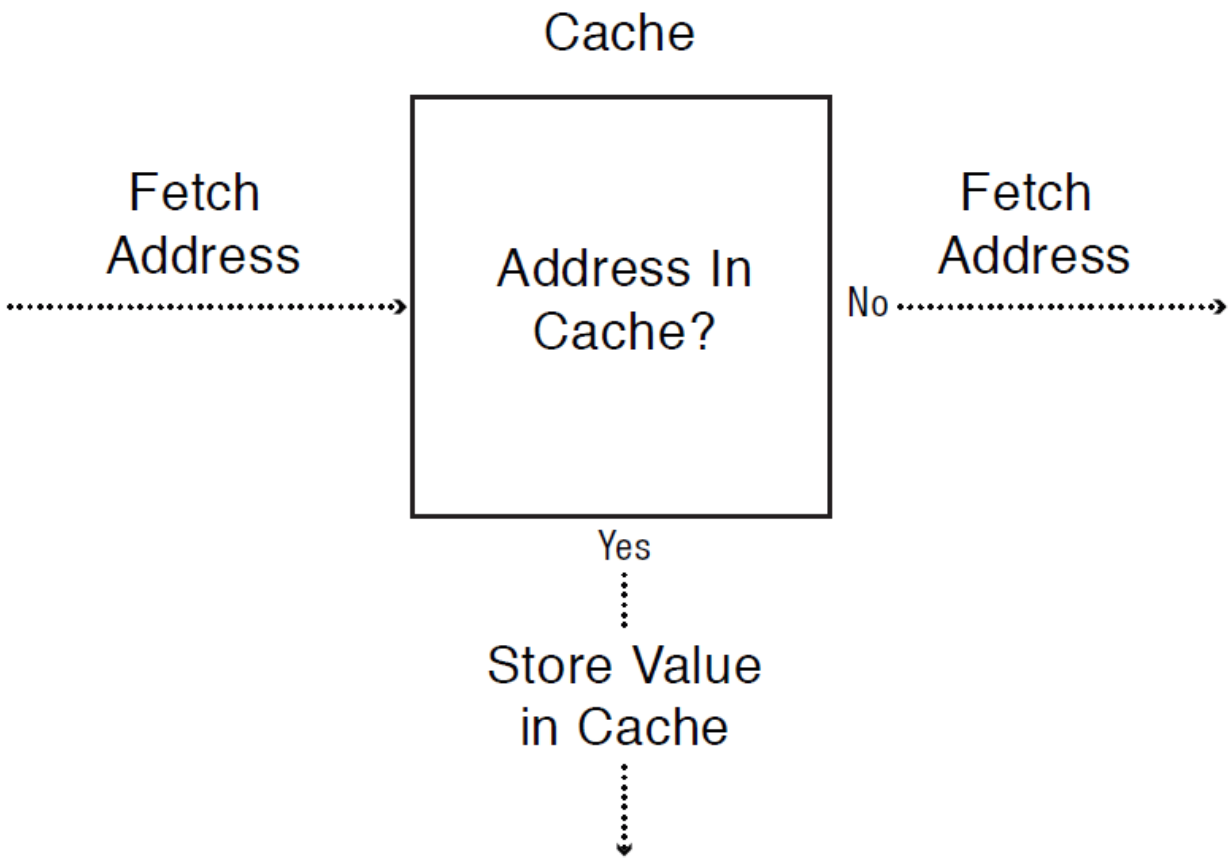
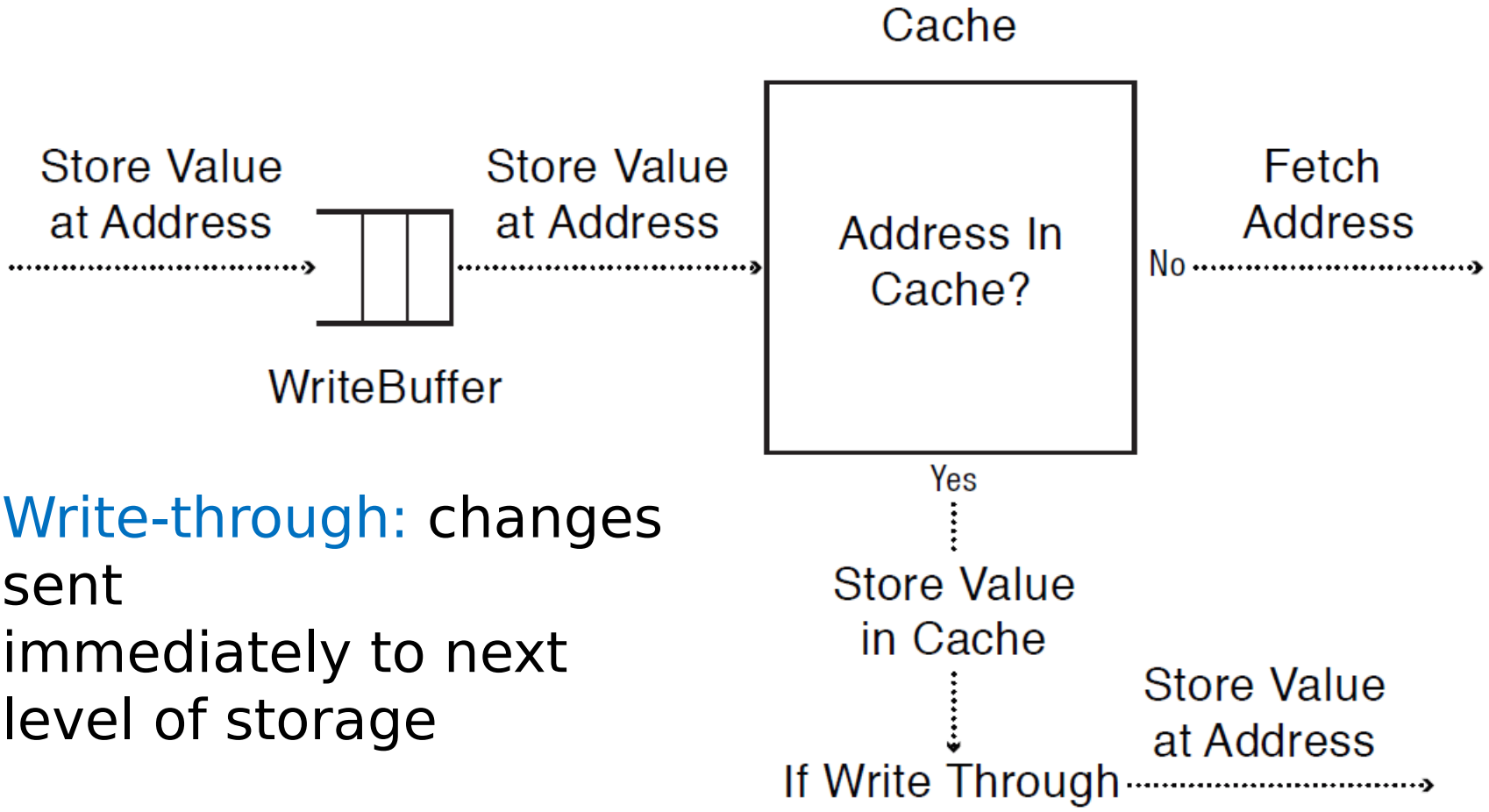


Diagram from Amir Kleen, et al., “Optimizing for instruction caches,” EE Times, Oct. 29, 2007, http://www.eetimes.com/document.asp?doc_id=1275470

Cache Concept (Read)



Cache Concept (Write)



Write-through: changes sent immediately to next level of storage

Write-back: changes stored in cache until cache block is replaced

Memory Hierarchy

Cache	Hit Cost	Size
1st level cache/first level TLB	1 ns	64 KB
2nd level cache/second level TLB	4 ns	256 KB
3rd level cache	12 ns	2 MB
Memory (DRAM)	100 ns	10 GB
Data center memory (DRAM)	100 μs	100 TB
Local non-volatile memory	100 μs	100 GB
Local disk	10 ms	1 TB
Data center disk	10 ms	100 PB
Remote data center disk	200 ms	1 XB

has 8MB as shared 3rd level cache; 2nd level cache is per-cor

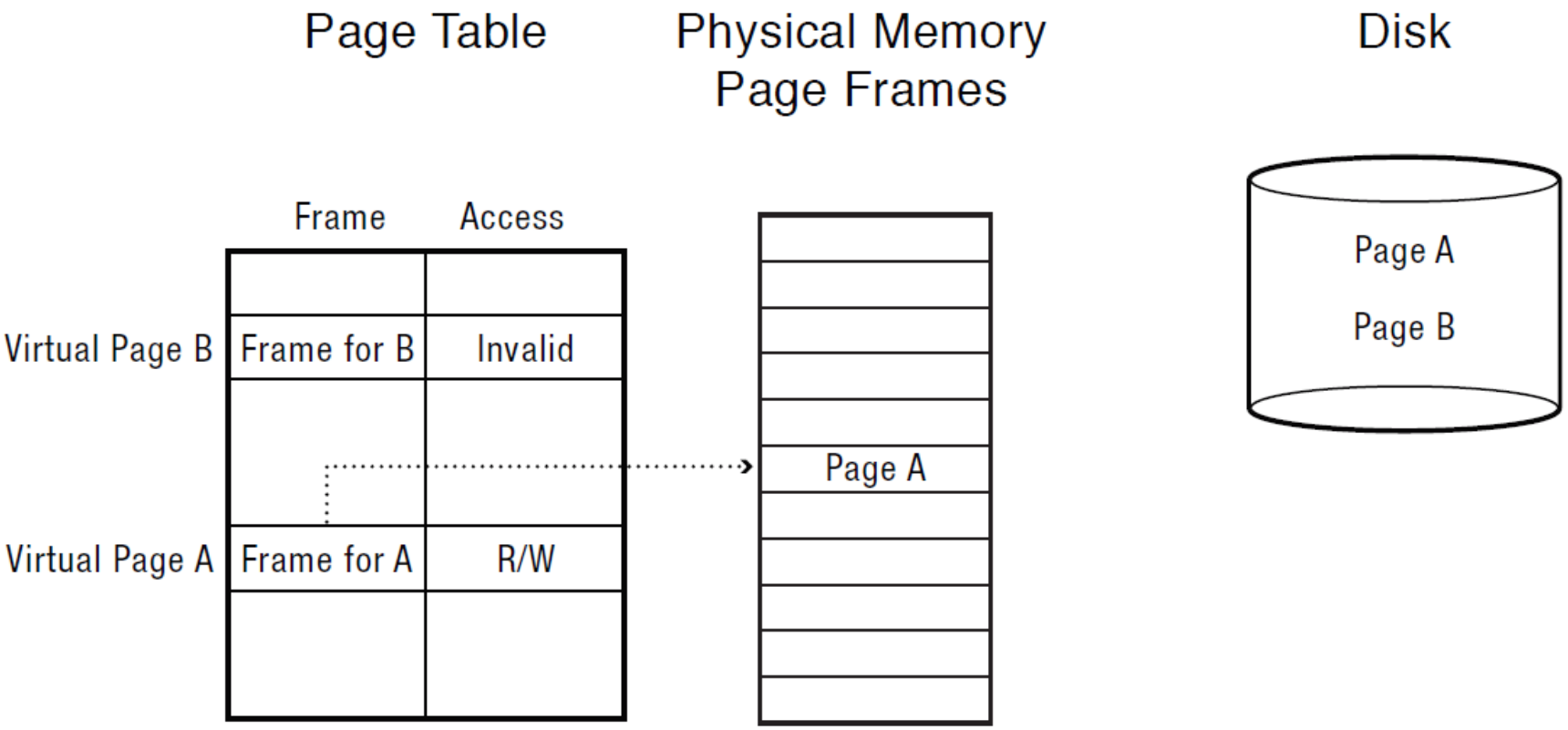
Main Points

- Can we provide the illusion of near infinite memory in limited physical memory?
  - Demand-paged virtual memory
  - Memory-mapped files
- How do we choose which page to replace?
  - FIFO, MIN, LRU, LFU, Clock
- What types of workloads does caching work for, and how well?
  - Spatial/temporal locality vs. Zipf workloads

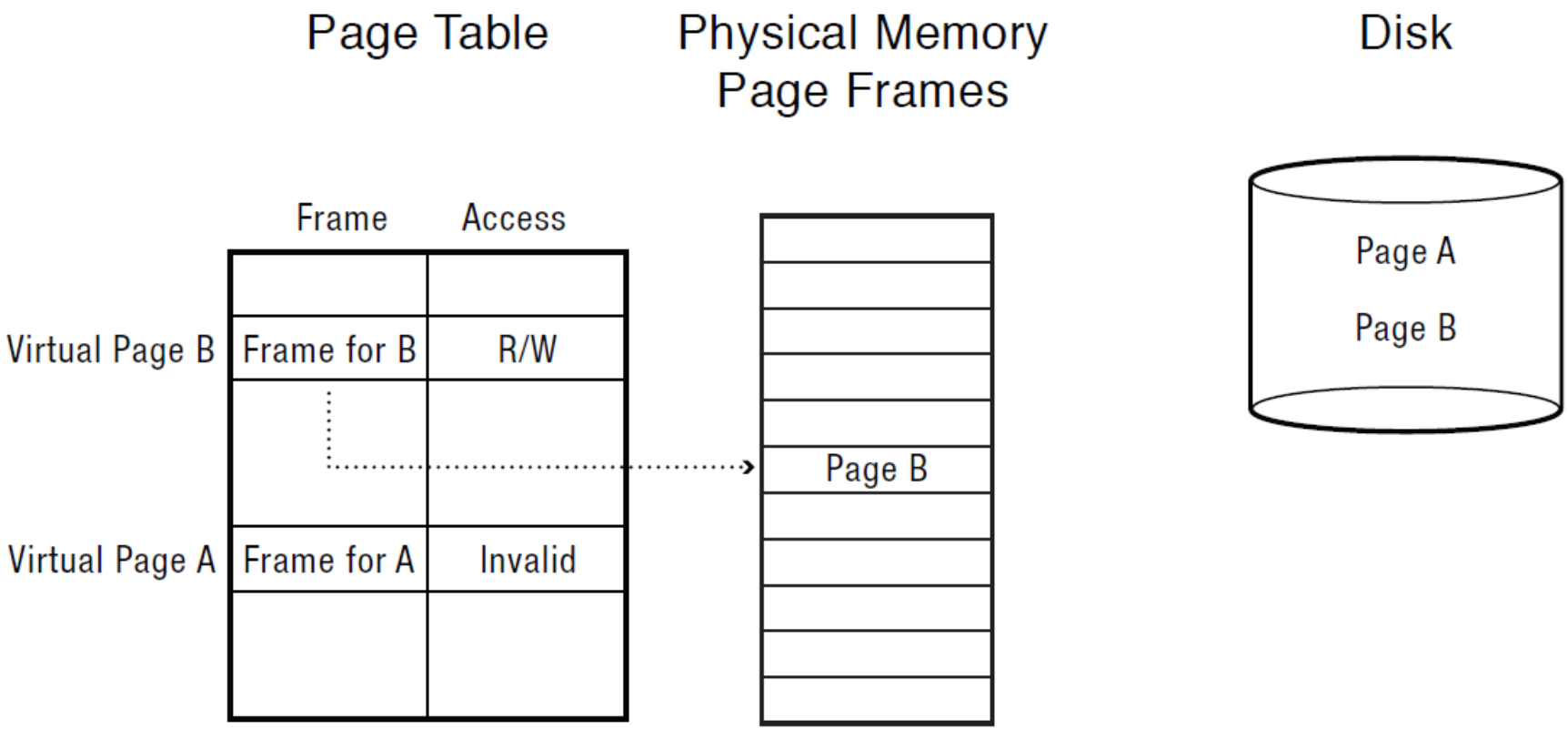
Hardware address translation is a power tool

- Kernel trap on read/write to selected addresses
  - Copy on write
  - Fill on reference
  - Zero on use
  - Demand paged virtual memory
  - Memory mapped files
  - Modified bit emulation
  - Use bit emulation

Demand Paging (Before)



Demand Paging (After)



Demand Paging

1. TLB miss and HW page table walk
2. Page fault (when page is marked as invalid / not present)

   --Traps to kernel--
3. Convert virtual address to file offset
4. Allocate page frame, initiate disk block read into page frame, and schedule another process or thread
5. Disk interrupt when DMA complete
6. Mark page as valid / present
7. Resume thread at faulting instruction
8. TLB miss
9. HW page table walk to fetch translation and load TLB

Demand Paging on MIPS

Remember that MIPS uses a software-loaded TLB, so:

1. TLB miss

   --Traps to kernel--
2. SW page table walk
3. Find page is marked as invalid / not present
4. Convert virtual address to file offset
5. Allocate page frame, initiate disk block read into page frame, and schedule another process or thread
6. Disk interrupt when DMA complete
7. Mark page as valid / present
8. Load TLB entry
9. Resume process at faulting instruction

Allocating a Page Frame

- Select page to evict
- Find all page table entries that refer to that page
  - Look at core map to see where page frame is shared
- Set each page table entry invalid / not present
- Remove any TLB entries
  - Copies of now invalid page table entry
- Write changes on page back to disk, if necessary

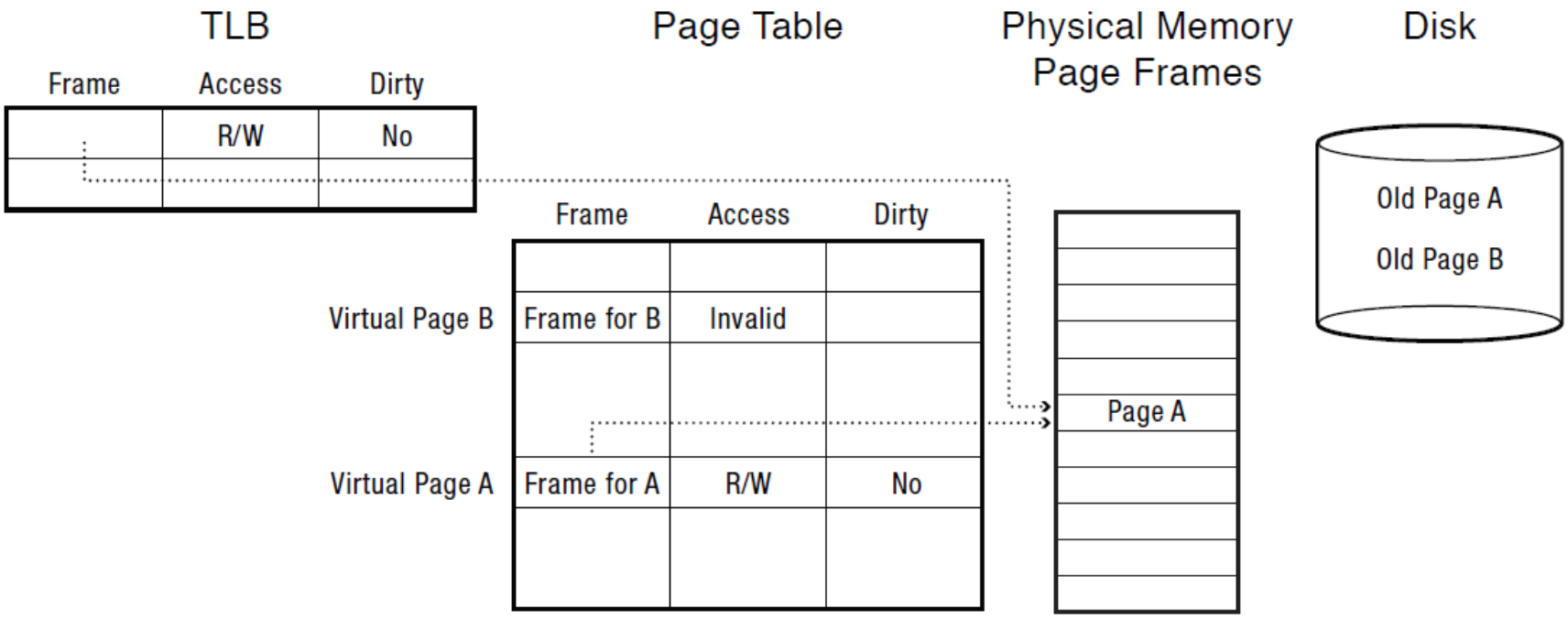
How do we know if page has been modified?

- Page table entries contain bookkeeping bits
  - Dirty bit: has page been modified?
    - Set by hardware on first store instruction to clean page
    - In both TLB and page table entry
  - Use bit: has page been recently used?
    - Set by hardware in page table entry on TLB miss
- Bookkeeping bits can be reset by the OS kernel
  - When changes to page are flushed to disk
  - To track whether page is recently used

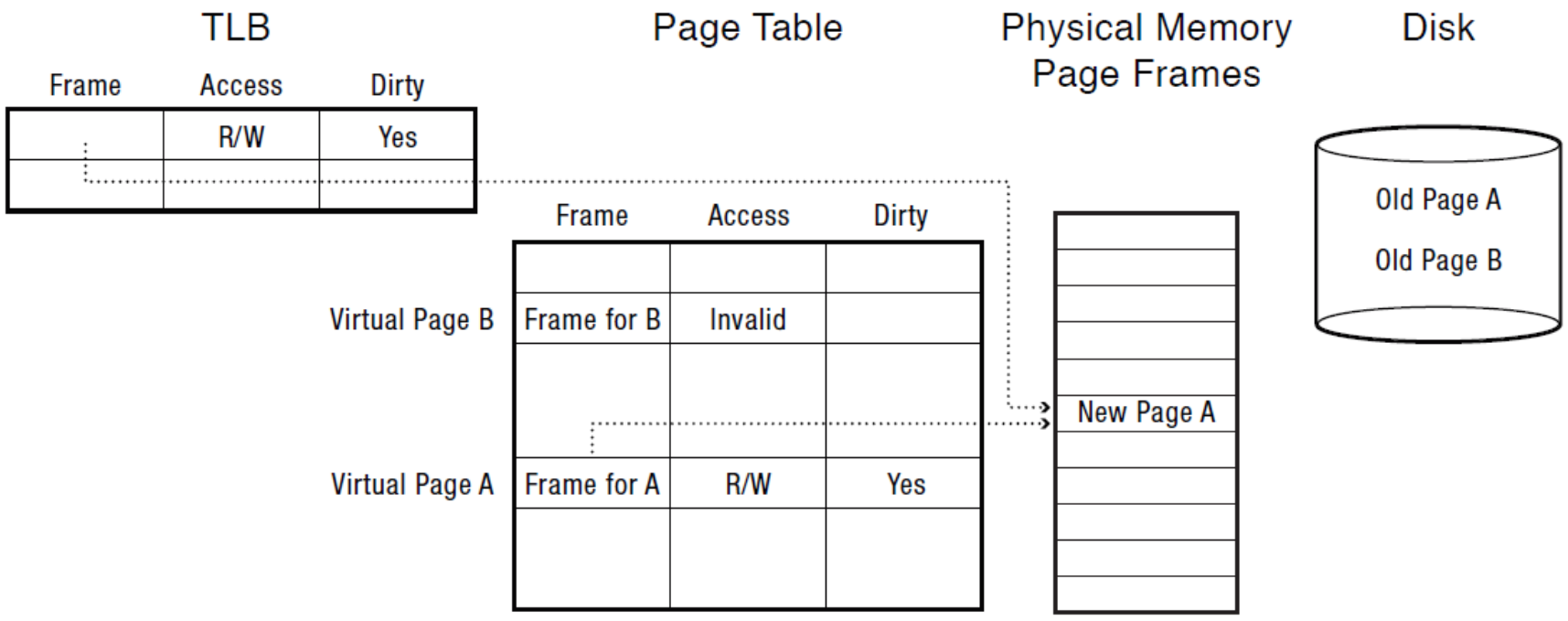
Terminology

- Different terms have been used by the processor manufacturers and OS programmers
- Modified == changed == dirty
- Recently used == use == accessed == reference

Keeping Track of Page Modifications (Before)



Keeping Track of Page Modifications (After)



Virtual or Physical Dirty/Use Bits

- Most machines keep dirty/use bits in the page table entry
- Physical page is
  - Modified if *any* page table entry that points to it is modified
  - Recently used if *any* page table entry that points to it is recently used
- On MIPS, simpler to keep dirty/use bits in the core map
  - Core map: map of physical page frames

Intel x86-32 Paging Control Bits

- P – presence
- R/W – read/write
  - Writes only allowed if bit is 1
- U/S – user/supervisor
  - User-mode accesses only allowed if bit is 1
- A – accessed (i.e., use)
- D – dirty
- XD – execution disabled
- G – global page
  - All processes share a global page using the same protection, e.g., kernel
- PWT – write-through (for cache control)
- PCD – cache disable (for cache control)
- PAT – page attribute table (for cache control)

Intel x86-32 Paging Control Bits (2)

- Presence and accessed bits are not needed in TLB entry for PTE
  - Why not?
- To correctly record page accesses and writes, a page's TLB entry must be invalidated when the OS resets the accessed bit or dirty bit in the PTE, and when the page is replaced
  - Why?

Models for Application File I/O

- Explicit read/write system calls
  - Data copied to user process using system call
  - Application operates on data
  - Data copied back to kernel using system call
- Memory-mapped files
  - Open file as a memory segment
  - Program uses load/store instructions on segment memory, implicitly operating on the file
  - Page fault if portion of file is not yet in memory
  - Kernel brings missing blocks into memory, restarts process

Advantages to Memory-mapped Files

- Programming simplicity, esp. for large files
  - Operate directly on file, instead of copy in/copy out
- Zero-copy I/O
  - Data brought from disk directly into page frame
- Pipelining
  - Process can start working before all the pages are populated
- Interprocess communication
  - Shared memory segment vs. temporary file

From Memory-Mapped Files to Demand-Paged Virtual Memory

- Every process segment backed by a file on disk
  - Code segment -> code portion of executable
  - Data, heap, stack segments -> temp files
  - Shared libraries -> code file and temp data file
  - Memory-mapped files -> memory-mapped files
  - When process ends, delete temp files
- Unified memory management across file buffer and process memory

Cache Replacement Policy

- On a cache miss, how do we choose which entry to replace?
  - Assuming the new entry is more likely to be used in the near future
  - In direct mapped caches, not an issue!

- Policy goal: reduce cache misses
  - Improve expected case performance
  - Also: reduce likelihood of very poor performance

A Simple Policy

- Random?
  - Replace a random entry
- FIFO?
  - Replace the entry that has been in the cache the longest time
  - What could go wrong?

FIFO in Action

FIFO

Reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			

Worst case for FIFO is if program strides through memory that is larger than the cache

MIN, LRU, LFU

- MIN
  - Replace the cache entry that will not be used for the longest time into the future
  - Optimality proof based on exchange: if evict an entry used sooner, that will trigger an earlier cache miss
- Least Recently Used (LRU)
  - Replace the cache entry that has not been used for the longest time in the past
  - Approximation of MIN
- Least Frequently Used (LFU)
  - Replace the cache entry used the least often (in the recent past)

LRU/MIN for Sequential Scan

							LRU								
Reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			
							MIN								
1	A					+					+			+	
2		B					+					+	C		
3			C					+	D					+	
4				D	E					+					+

LRU

Reference	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A		+				+				+			+	
2		B			+								+		
3				C					E			+			
4						D		+		+					C

FIFO

1	A		+				+		E						
2		B			+						A			+	
3				C								+	B		
4						D		+		+					C

MIN

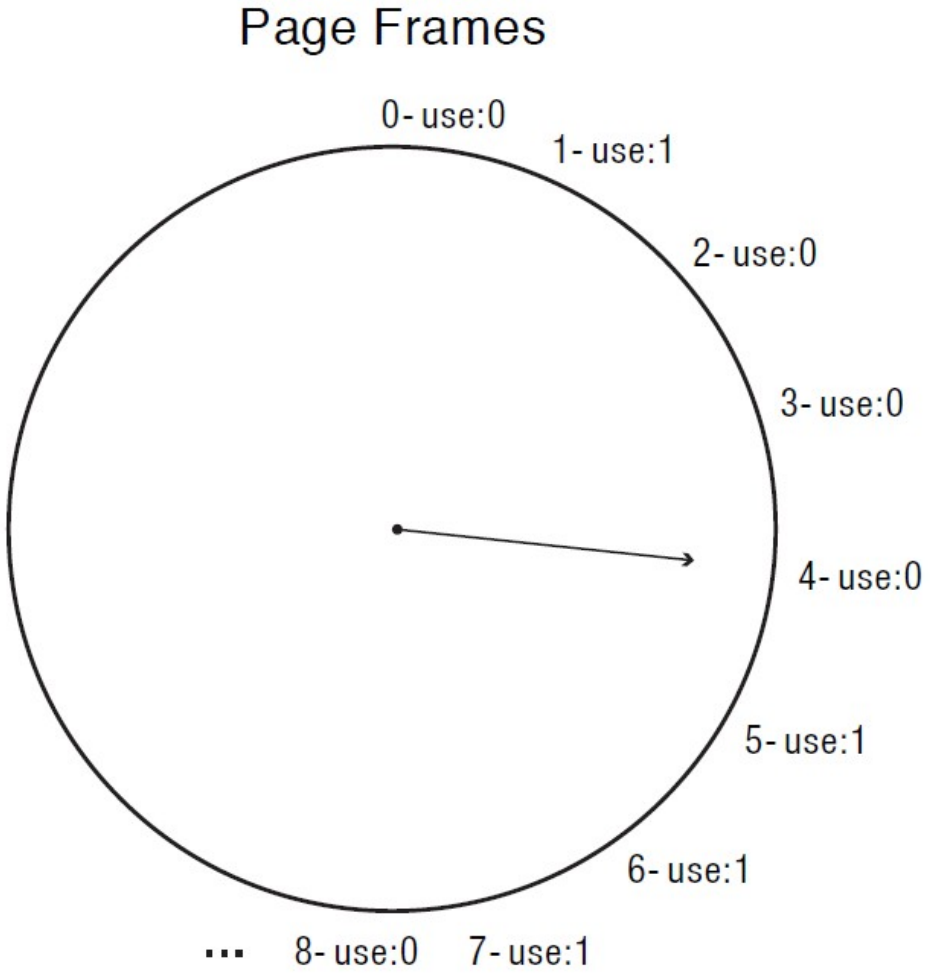
1	A		+				+				+			+	
2		B			+								+		C
3				C					E			+			
4						D		+		+					

Belady's Anomaly

FIFO (3 slots)												
Reference	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					+
2		B			A			+		C		
3			C			B			+		D	
FIFO (4 slots)												
1	A				+		E				D	
2		B				+		A				E
3			C						B			
4				D						C		

Clock Algorithm: Estimating LRU

- Periodically, sweep through all pages
- If page is unused, reclaim
- If page is used, mark as unused



Nth Chance: Not Recently Used

- Instead of one bit per page, keep an integer
  - notInUseSince: number of sweeps since last use
- Periodically sweep through all page frames

```
if (page is used) {
    notInUseSince = 0;
} else if (notInUseSince < N) {
   notInUseSince++;
} else {
     reclaim page;
}
```

Implementation Note

- Clock and Nth Chance can run synchronously
  - In page fault handler, run algorithm to find next page to evict
  - Might require writing changes back to disk first
- Or asynchronously
  - Create a kernel thread to maintain a pool of recently unused, clean pages
  - Find recently unused dirty pages, write mods back to disk
  - Find recently unused clean pages, mark as invalid / not present and move to pool
  - On page fault, check if requested page is in pool! (like VAX/VMS victim buffer)
  - If not, evict that page

DEC VAX/VMS Fast Reclaim

- VAX-11/780 did not provide use bits
- OS designers used FIFO with victim buffer replacement policy
  - Oldest pages in FIFO list placed into a victim buffer
  - Victim pages are still in memory but unmapped
  - A page fault to a victim page allowed fast reclaim, i.e., no disk access
- Performance almost as good as LRU

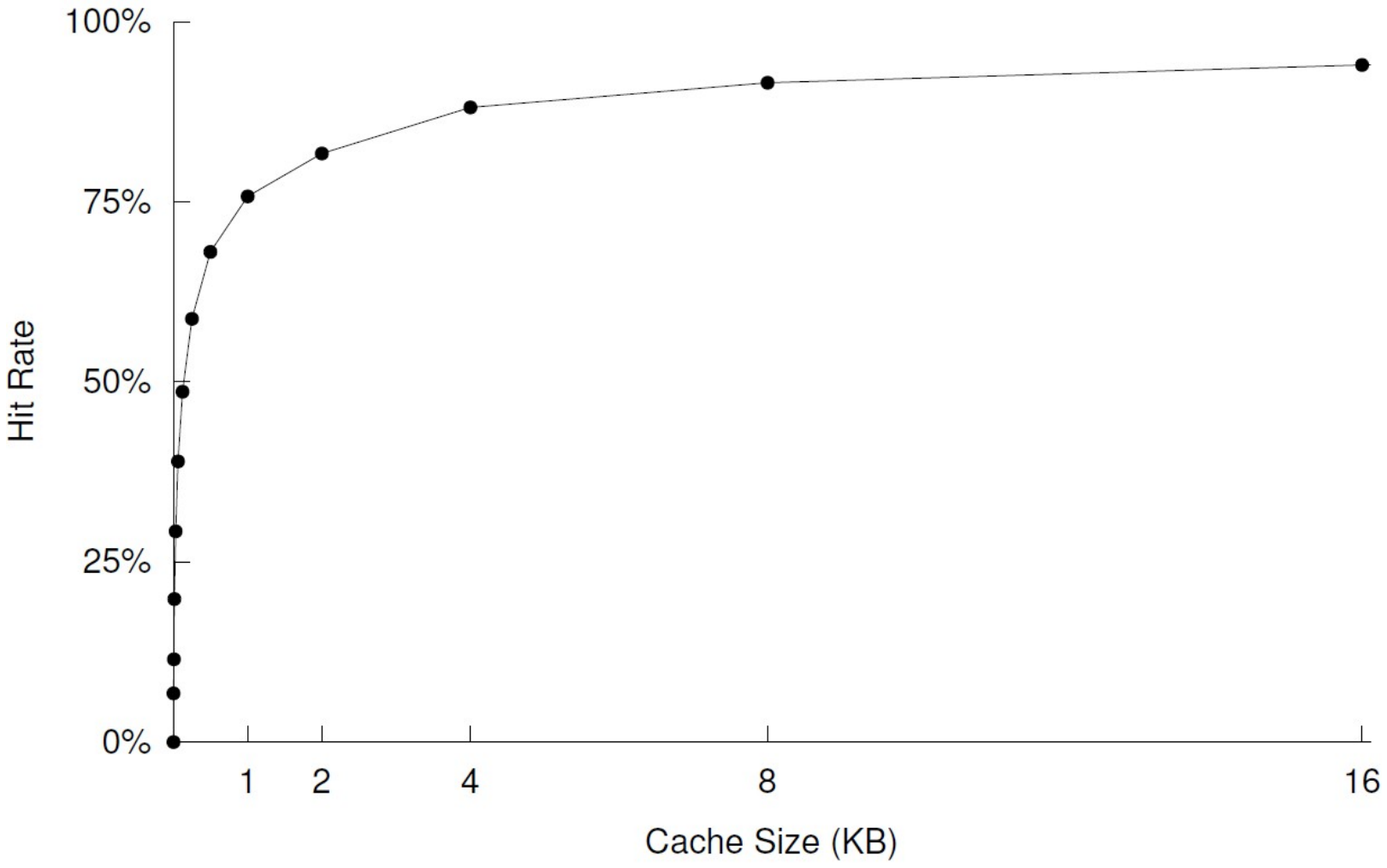
Recap

- MIN is optimal
  - Replace a page or cache entry that will not be used again or the one used farthest into the future
- LRU is an approximation of MIN
  - For programs that exhibit spatial and temporal locality
- Clock/Nth Chance is an approximation of LRU
  - Can bin pages into sets of “not recently used”

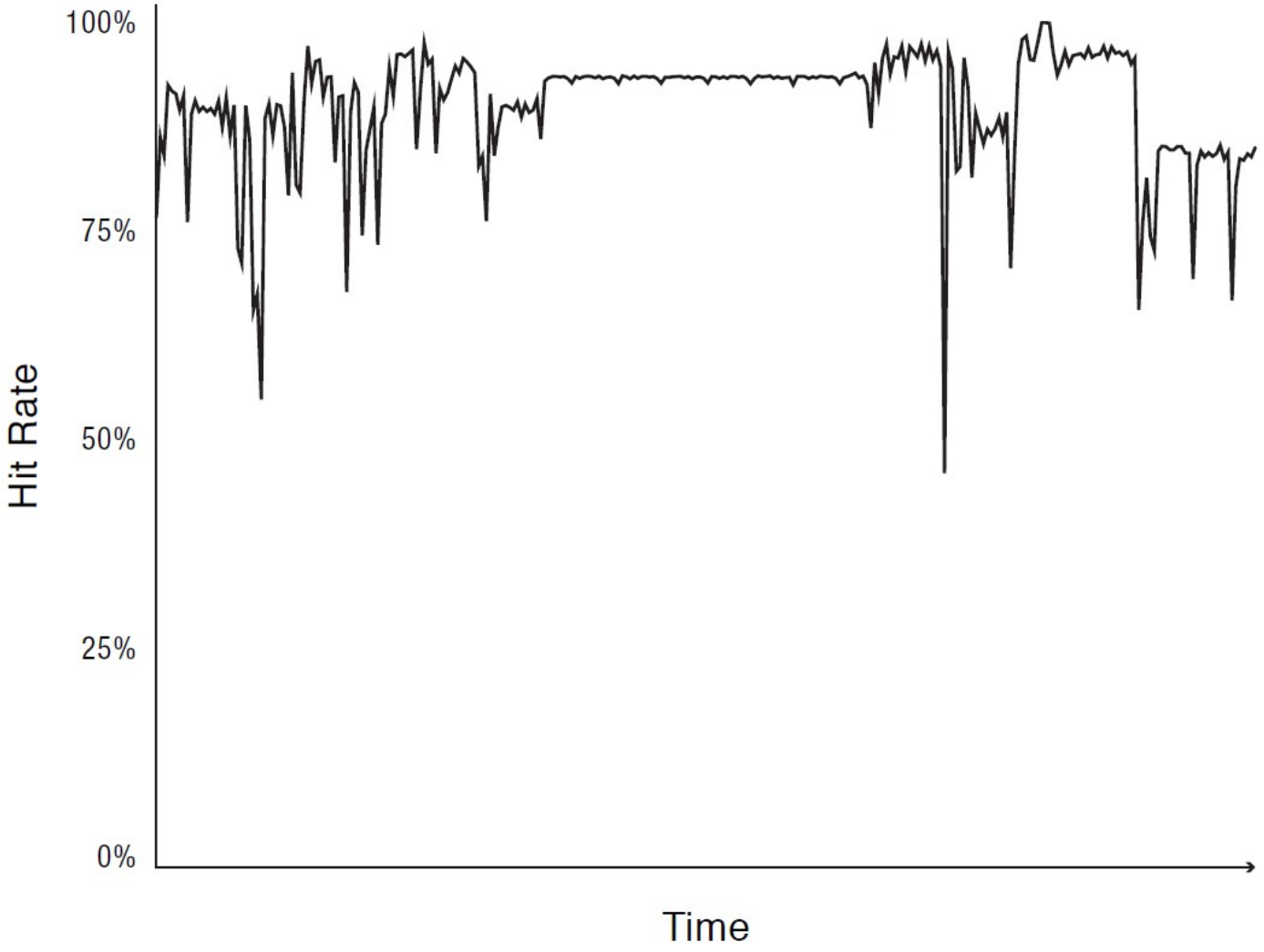
Working Set Model

- Working Set: set of memory locations that need to be cached for reasonable cache hit rate
- Thrashing: when system has too small a cache

Cache Working Set



Phase Change Behavior



Question

- What happens to system performance as we increase the number of processes?
  - If the sum of the working sets > physical memory?

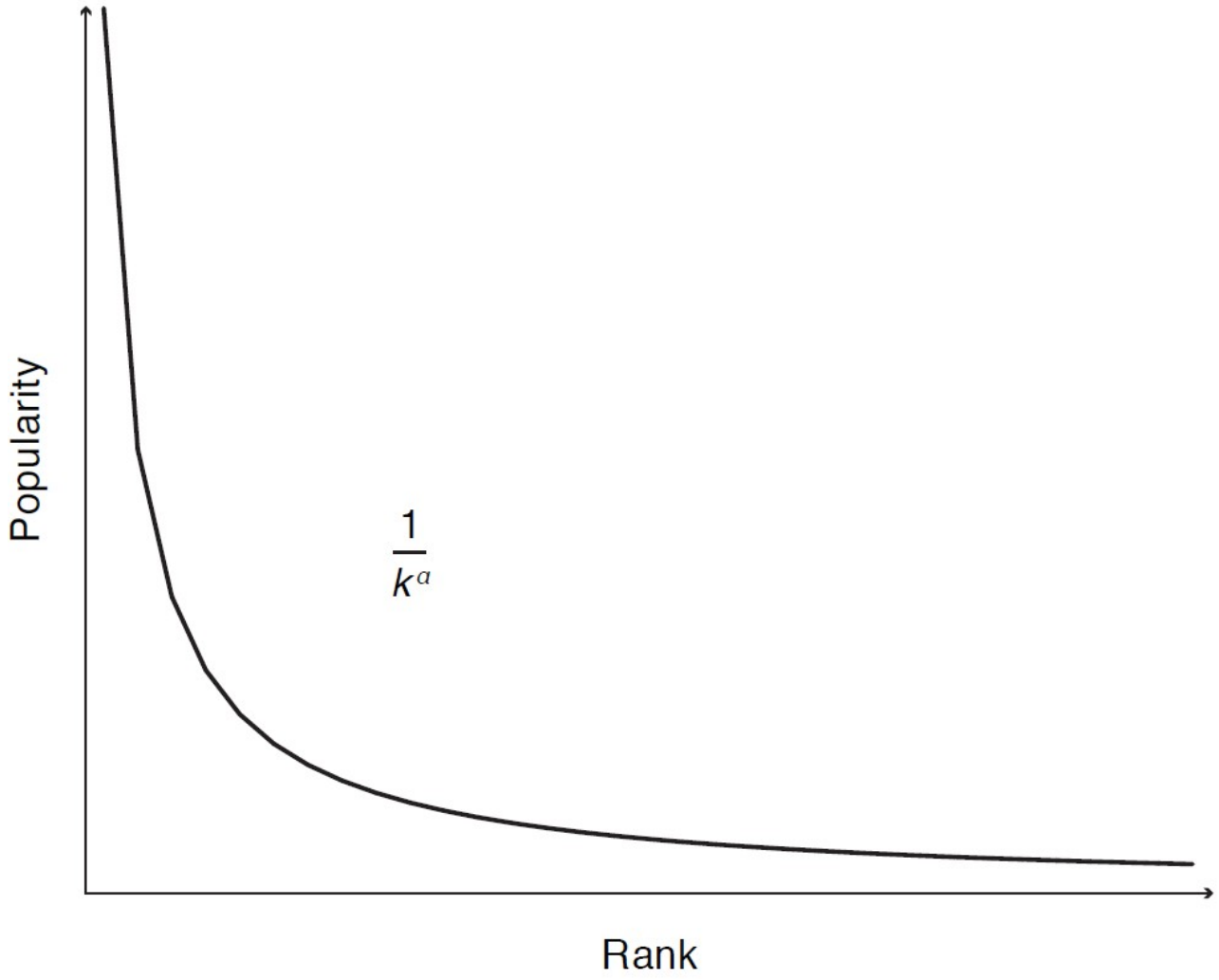
Prevent Over-Commitment

- Working Set (WS) page allocation policy
  - Track working sets
  - If sum of working sets is too large then suspend a process
  - Can use a periodic inquiry and reset of the use bits to determine working set membership
- Page Fault Frequency (PFF) page allocation policy
  - Track page fault rate of each process
  - If the rate for a process goes above a "high-water" threshold then steal a page from a process with a rate below a "low-water" threshold
  - If there are no pages available to steal then suspend a process
  - No need for use bits to implement this policy

Zipf Distribution

- Caching behavior of many systems are not well characterized by the working set model
- An alternative is the Zipf distribution
  - Popularity ~ $1/k^{\alpha}$, for k^{th} most popular item, $1 < \alpha < 2$

Zipf Distribution

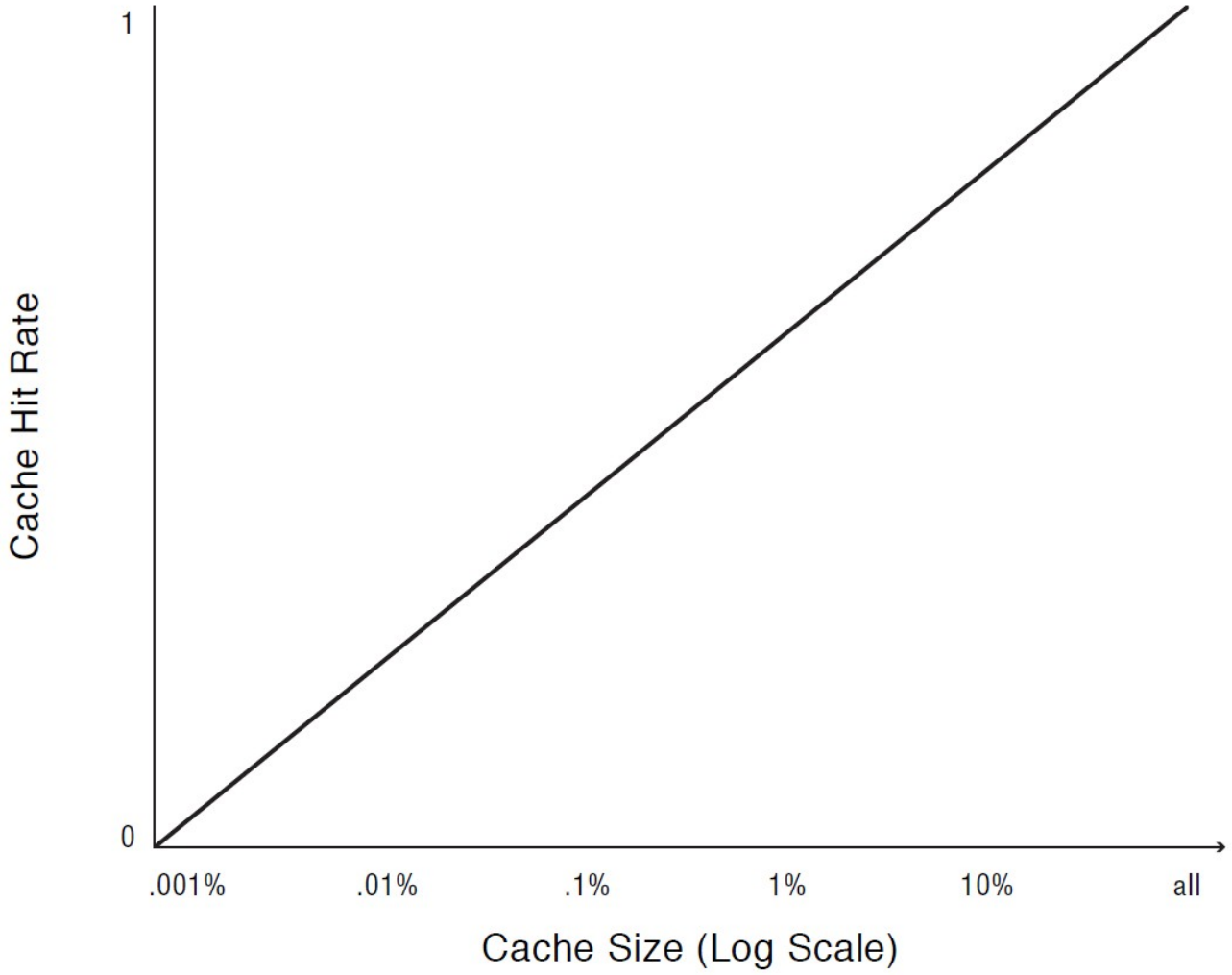


Zipf Examples

- Web pages
- Movies
- Library books
- Words in text
- Salaries
- City population
- ...

Common thread: popularity is self-reinforcing

Zipf and Caching



(if time permits)

Emulating Modified/Use Bits w/ MIPS Software-Loaded TLB

- MIPS TLB entries have an extra bit: modified/unmodified
  - Trap to kernel if no entry in TLB, or if write to an unmodified page
- On a TLB read miss:
  - If page is clean, load TLB entry as read-only; if dirty, load as rd/wr
  - Mark page as recently used
- On a TLB write to an unmodified page:
  - Kernel marks page as modified in its page table
  - Reset TLB entry to be read-write
  - Mark page as recently used
- On TLB write miss:
  - Kernel marks page as modified in its page table
  - Load TLB entry as read-write
  - Mark page as recently used

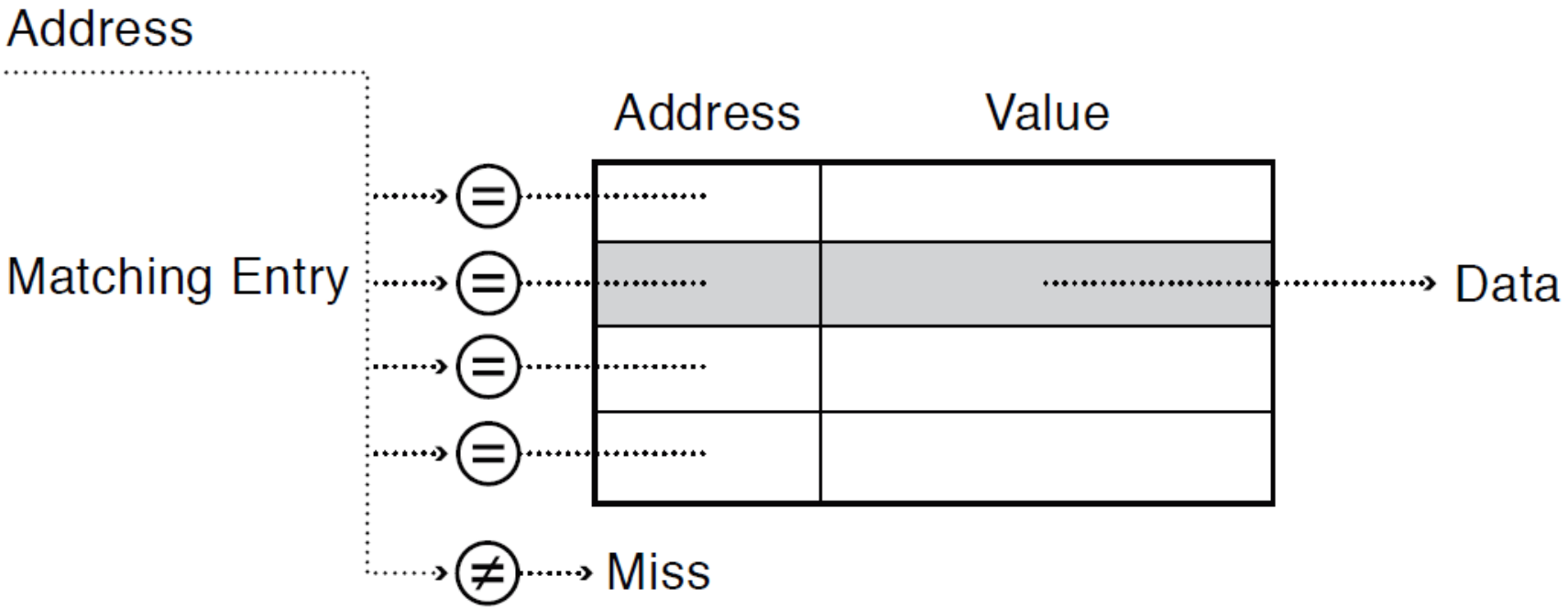
Emulating a Modified Bit (Hardware-Loaded TLB)

- Some processor architectures do not keep a modified bit per page
  - Extra bookkeeping and complexity
- Kernel can emulate a modified bit:
  - Set all clean pages as read-only
  - On first write to page, trap into kernel
  - Kernel sets modified bit, marks page as read-write
  - Resume execution
- Kernel needs to keep track of both
  - Current page table permission (e.g., read-only)
  - True page table permission (e.g., writeable, clean)

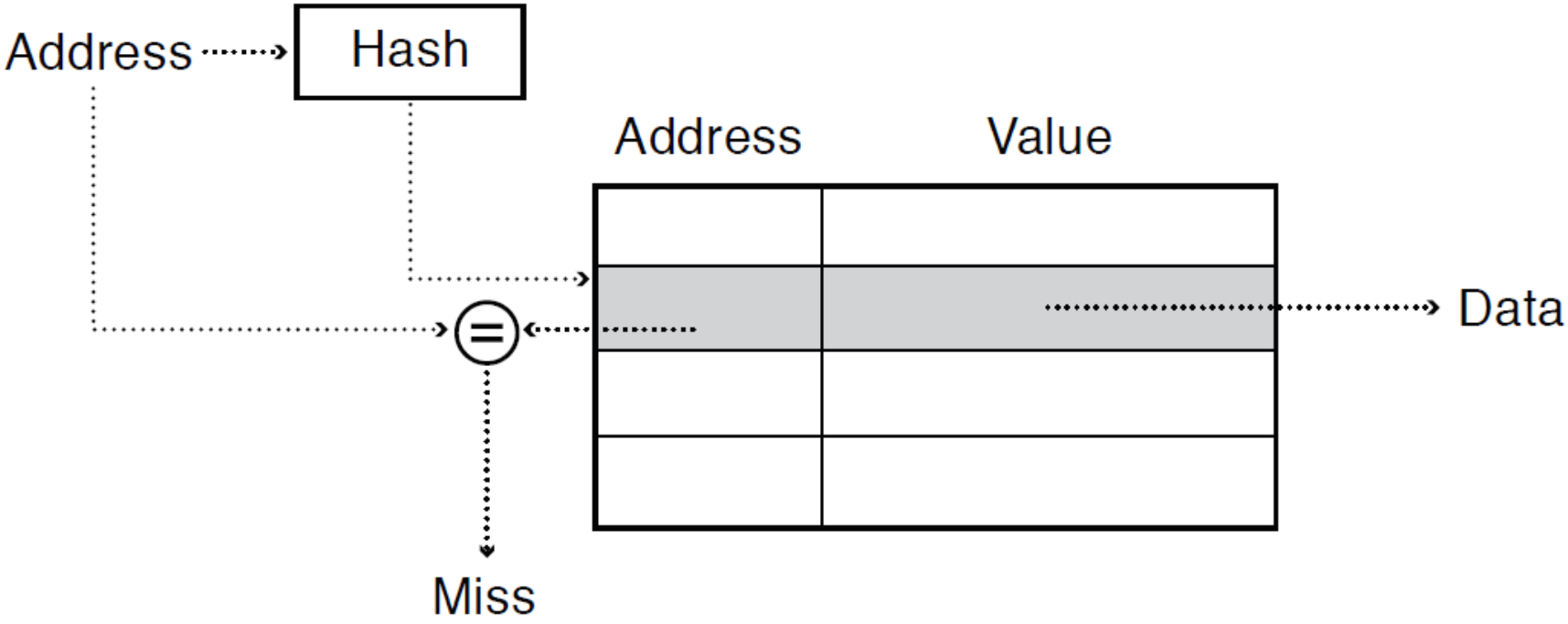
Emulating a Recently Used Bit (Hardware-Loaded TLB)

- Some processor architectures do not keep a recently used bit per page
  - Extra bookkeeping and complexity
- Kernel can emulate a recently used bit:
  - Set all recently unused pages as invalid / not present
  - On first read/write, trap into kernel
  - Kernel sets recently used bit
  - Marks page as read or read/write
- Kernel needs to keep track of both
  - Current page table permission (e.g., invalid / not present)
  - True page table permission (e.g., read-only, writeable)

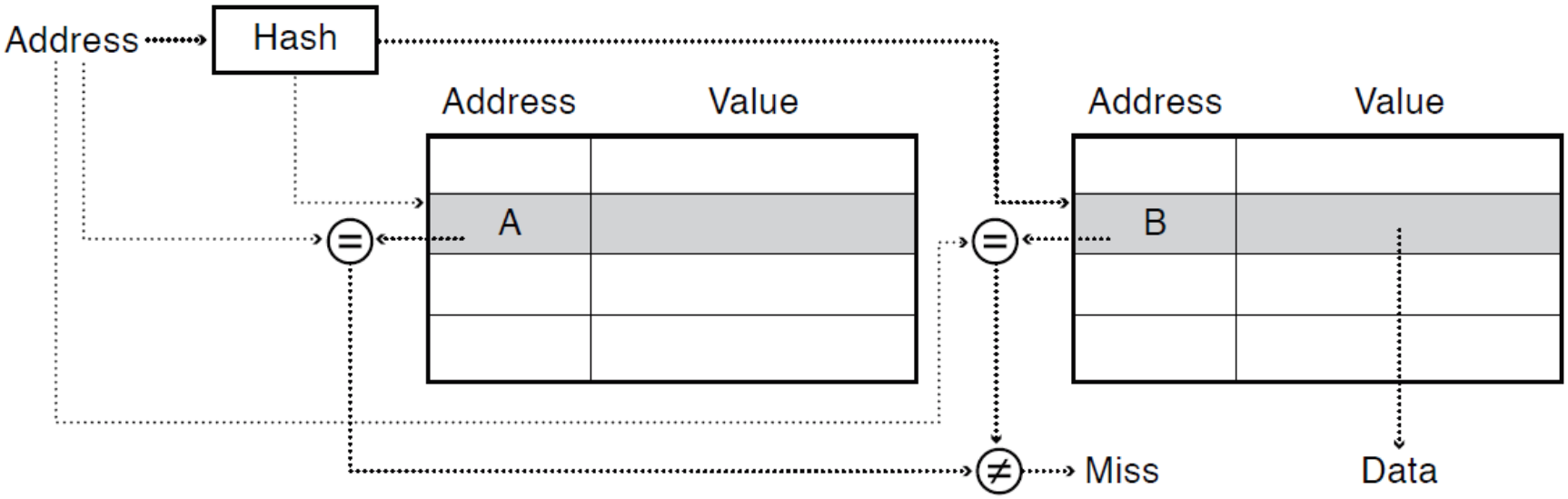
Cache Lookup: Fully Associative



Cache Lookup: Direct Mapped



Cache Lookup: Set Associative



Page Coloring

- What happens when cache size >> page size?
  - Direct mapped or set associative
  - Multiple pages map to the same cache line
- OS page assignment matters!
  - Example: 8MB cache, 4KB pages
  - 1 of every 2K pages lands in same place in cache
- What should the OS do?

Page Coloring