

# Computer Architecture: Cache Design: Part Two

Hossein Asadi ([asadi@sharif.edu](mailto:asadi@sharif.edu))

Department of Computer Engineering

Sharif University of Technology

Spring 2024



# Copyright Notice

---

- Some Parts (text & figures) of this Lecture adopted from following:
  - D.A. Patterson and J.L. Hennessy, “[Computer Organization and Design: the Hardware/Software Interface](#)” (MIPS), 6<sup>th</sup> Edition, 2020.
  - J.L. Hennessy and D.A. Patterson, “[Computer Architecture: A Quantitative Approach](#)”, 6<sup>th</sup> Edition, Nov. 2017.
  - “Intro to Computer Architecture” handouts, by Prof. Hoe, CMU, Spring 2009.
  - “Computer Architecture & Engineering” handouts, by Prof. Kubiawicz, UC Berkeley, Spring 2004.
  - “Intro to Computer Architecture” handouts, by Prof. Hoe, UWisc, Spring 2021.
  - “Computer Arch I” handouts, by Prof. Garzarán, UIUC, Spring 2009.



# Topics Covered in This Lecture

---

- **Improving Cache Performance**
- **Sources of Cache Misses**
- **Reducing Miss Penalty**
- **Prefetching**
- **Cache Coherency**



# Reminder: Improving Cache Performance

---

- AMAT =  
Hit Time + (Miss Rate x Miss Penalty)
- Options to Reduce AMAT
  - Reduce time to hit in cache
    - Use smaller cache size
  - Reduce miss rate
    - Increase cache size
  - Reduce miss penalty
    - Use multi-level cache hierarchy



# Cache Hit Time

---

- Impact on Cycle Time
  - Directly tied to clock rate
  - Increases with cache size
  - Increases with associativity



# Sources of Cache Misses

---

- 3Cs
  - Compulsory
  - Capacity
  - Conflict
- Another source of cache miss
  - Coherence



# Sources of Cache Misses

---

- **Compulsory**

- Cold start or process migration
- First access to a block
- Compulsory misses are insignificant
  - When running “billions” of instruction

- **Capacity**

- Cache cannot contain all blocks accessed by program
- **Solution**: increase cache size



# Sources of Cache Misses

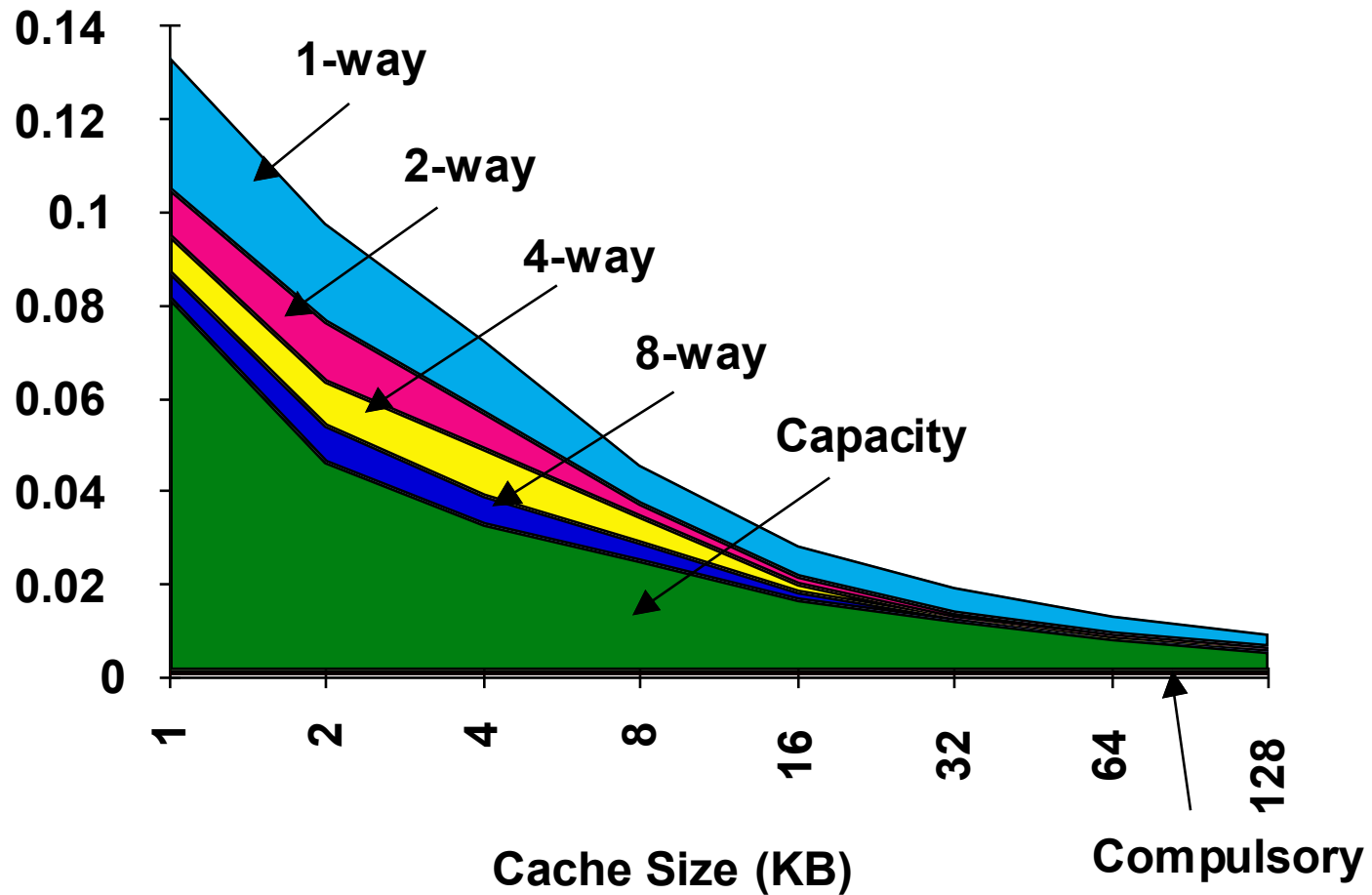
---

- **Conflict (collision)**
  - Multiple memory locations mapped to same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- **Coherence (Invalidation)**
  - Other processes (e.g., I/O or a core in a CMP) updates memory

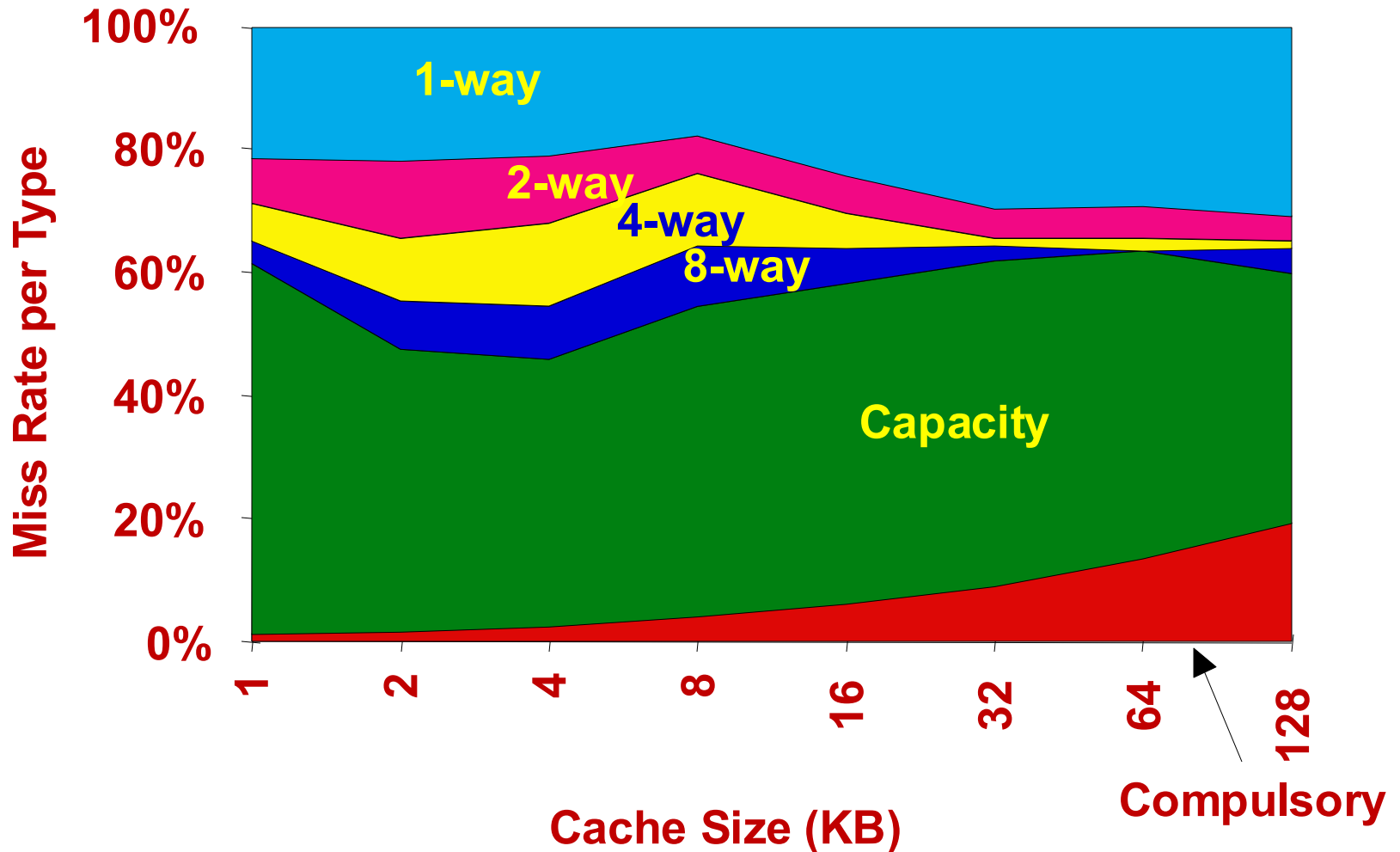




# 3Cs Absolute Miss Rate



# 3Cs Relative Miss Rate



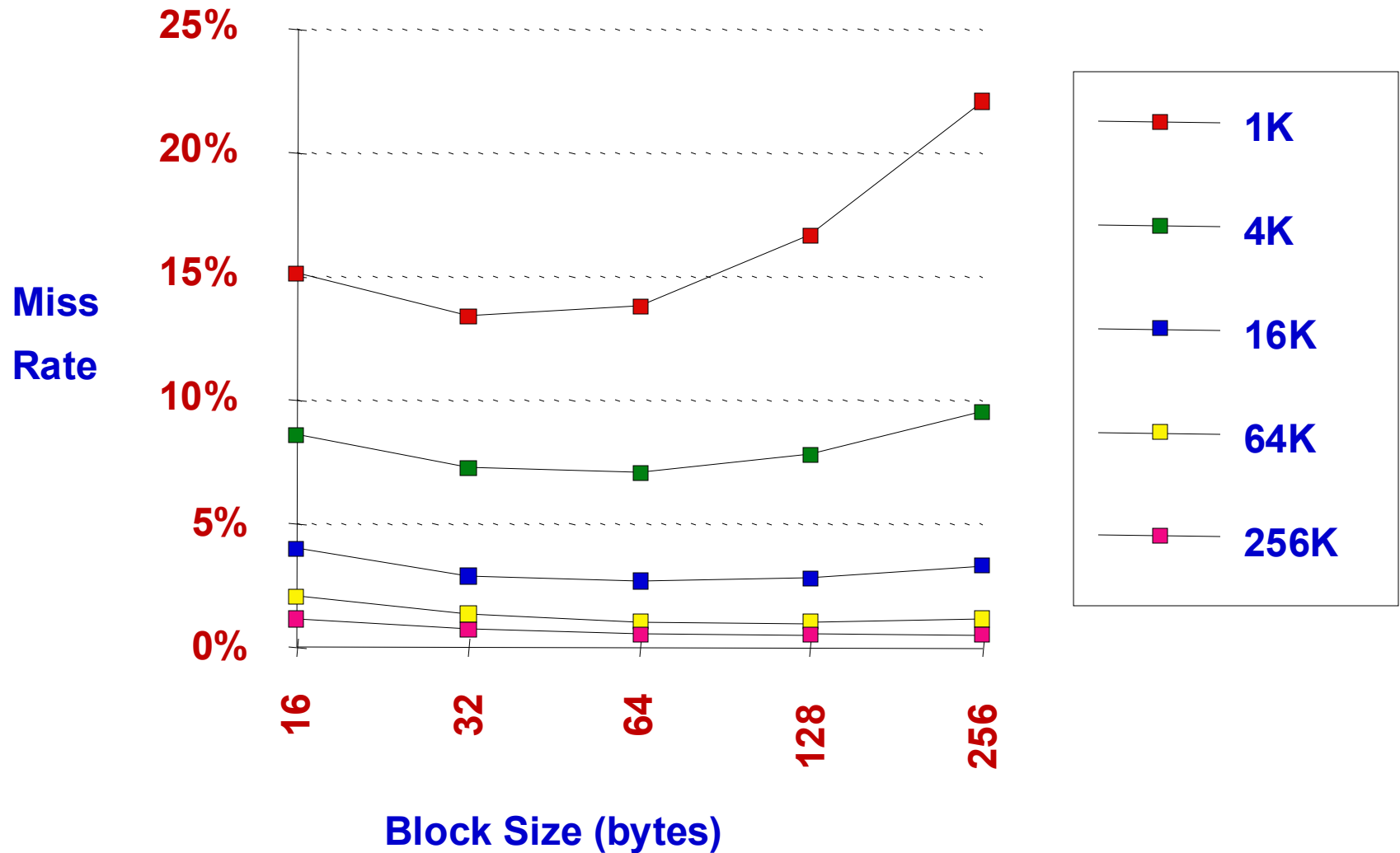
# Reducing Miss Rate

---

- Larger Block Size
- Higher Associativity
- Prefetching
- Compiler Optimization



# Reducing Misses via Larger Block Size



# Reducing Misses via Higher Associativity

- 2:1 Cache Rule:
  - Miss Rate DM cache size  $N$  = Miss Rate 2-way cache size  $N/2$
- Watch Out
  - Execution time is only final measure!
  - AMAT not always improved by more associativity!



# Reducing Misses by Prefetching

---

- Instruction Prefetching
- Data Prefetching
- HW vs. SW Prefetching



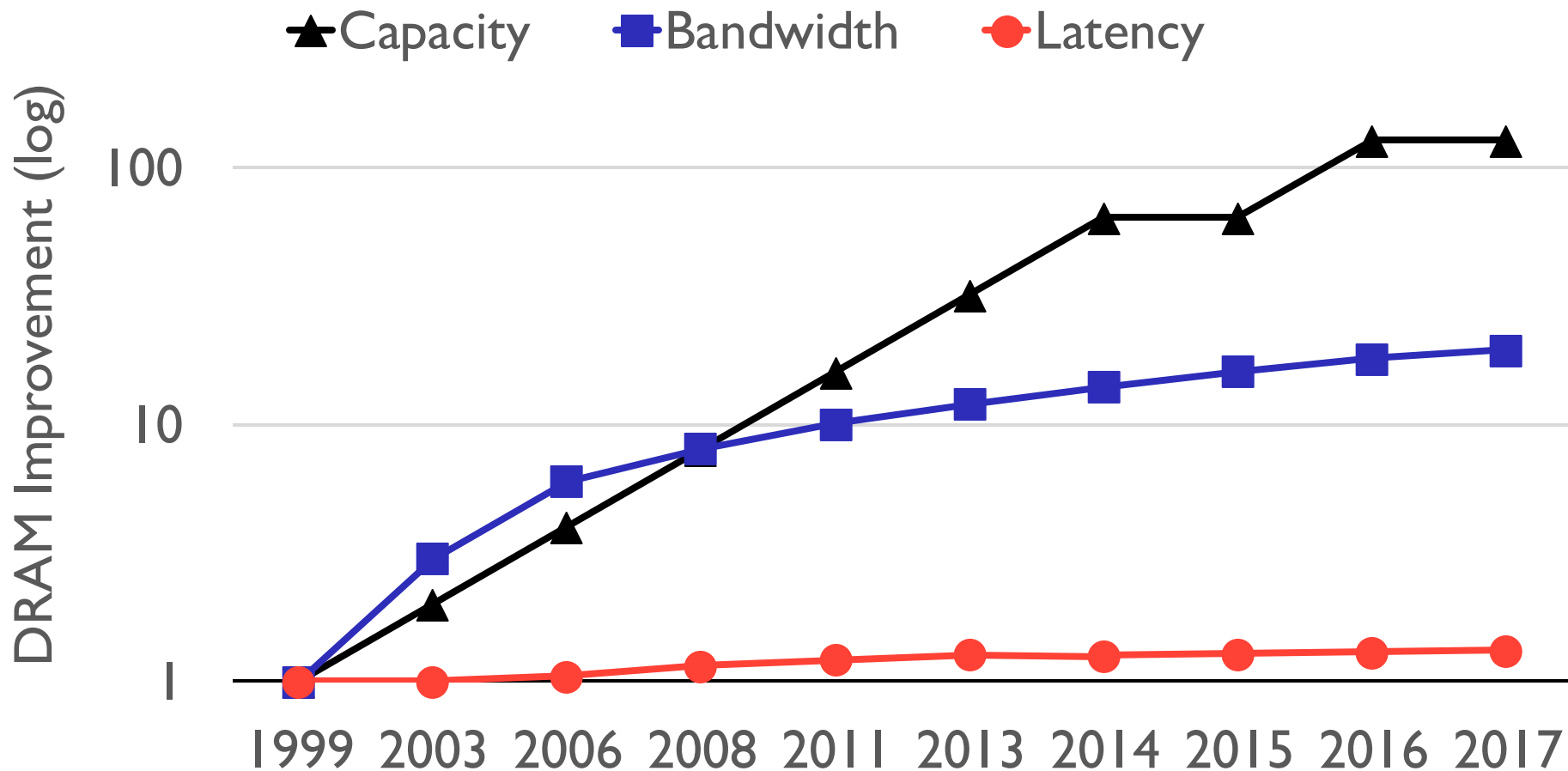
# Reducing Miss Penalty

---

- Faster RAM Technologies
  - Use of faster SRAMs and DRAMs
- More Hierarchy Levels
  - 1-level → 2-level → 3-level
- Read Priority over Write on Miss
  - Reads on critical path



# Why Prefetching?



Memory latency remains almost constant





# Prefetching

---

- Idea: **Fetch data before needed (i.e. pre-fetch) by the program**
- Why?
  - Memory latency is high. If we can prefetch **accurately** and **early enough**, we can reduce/eliminate that latency.
  - Can eliminate **compulsory cache misses**
  - Can it eliminate all cache misses? Capacity, conflict? Coherence?
- Involves predicting **which address** will be needed in the future
  - Works if programs have **predictable miss address patterns**



# Prefetching and Correctness

---

- Does a misprediction in prefetching affect correctness?
- No, prefetched data at a “mispredicted” address is simply not used
- There is no need for state recovery
  - In contrast to branch misprediction or value misprediction



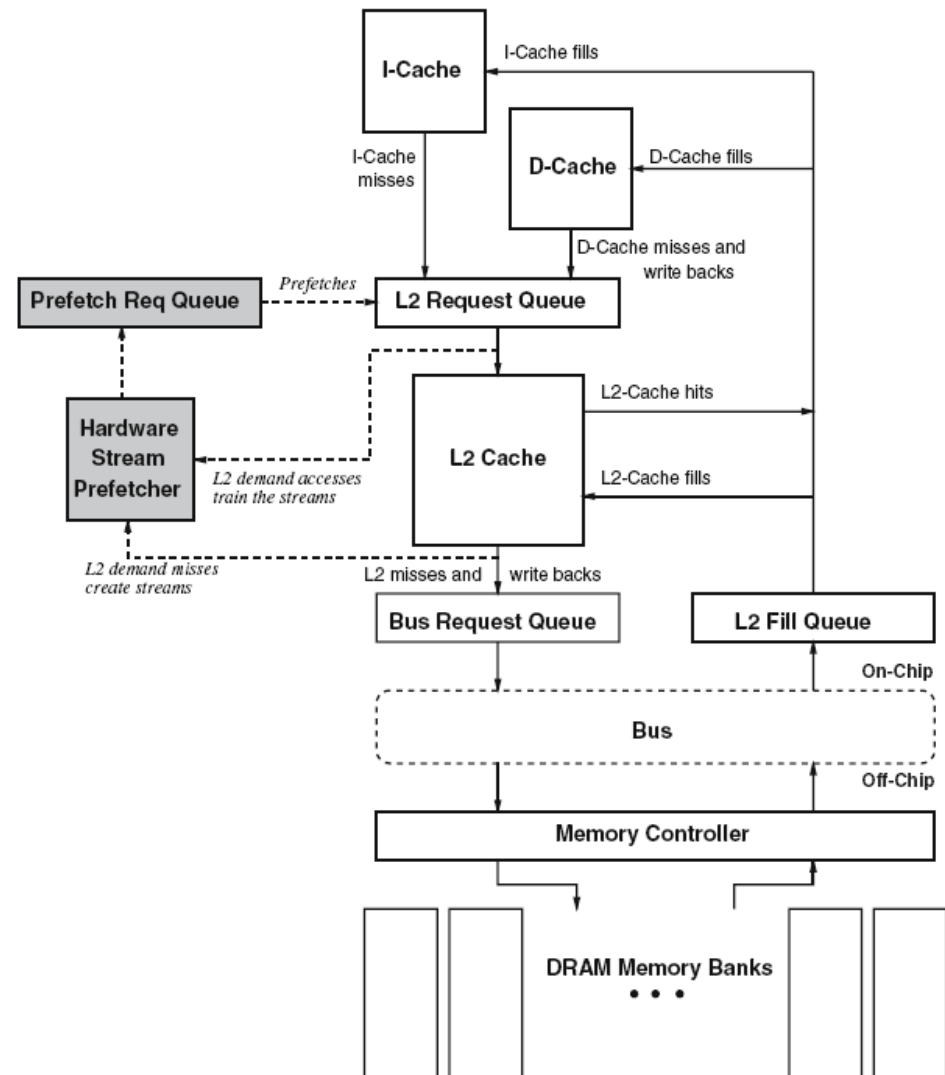
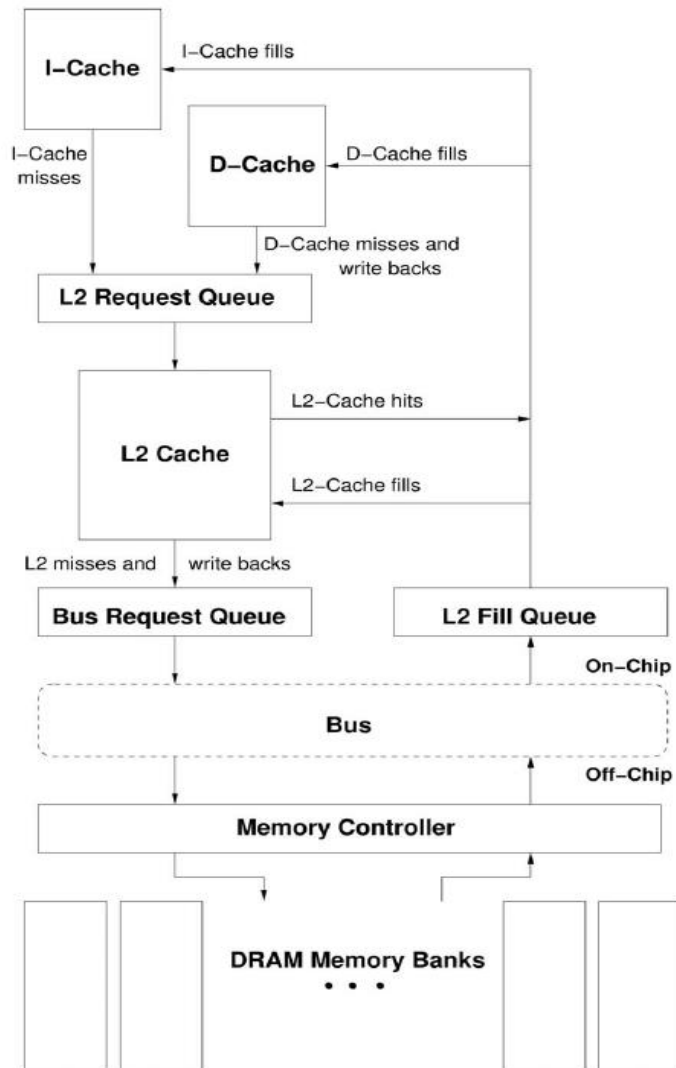
# Prefetching: Basics

---

- In modern systems, prefetching is usually done at **cache block granularity**
- Prefetching is a technique that can reduce both
  - Miss rate
  - Miss latency
- Prefetching can be done by:
  - Hardware
  - Compiler
  - Programmer
  - System



# How a HW Prefetcher Fits in the Memory System?



# Prefetching: The Four Questions

---

- What

- What addresses to prefetch (i.e., address prediction algorithm)

- When

- When to initiate a prefetch request (early, late, on time)

- Where

- Where to place the prefetched data (caches, separate buffer)
- Where to place the prefetcher (which level in memory hierarchy)

- How

- How does the prefetcher operate and who operates it (software, hardware, execution/thread-based, cooperative, hybrid)



# Challenge in Prefetching: What

---

- **What** addresses to prefetch
  - Prefetching **useless** data wastes resources
    - Memory bandwidth
    - Cache or prefetch buffer space
    - Energy consumption
    - These could all be utilized by demand requests or more accurate prefetch requests
  - **Accurate** prediction of addresses to prefetch is important
    - Prefetch accuracy = used prefetches / sent prefetches
- **How do we know what to prefetch?**
  - Predict based on past access patterns
  - Use the compiler's/programmer's knowledge of data structures
- **Prefetching algorithm** determines what to prefetch



# Challenges in Prefetching: When

---

- **When** to initiate a prefetch request
  - Prefetching too early
    - Prefetched data might not be used before it is evicted from storage
  - Prefetching too late
    - Might not hide the whole memory latency
- When a data item is prefetched affects the **timeliness** of the prefetcher
- Prefetcher can be made more timely by
  - Making it more **aggressive**: try to stay far ahead of the processor's demand access stream (hardware)
  - Moving the **prefetch instructions earlier in the code** (software)



# Challenges in Prefetching: Where (I)

---

- **Where** to place the prefetched data
  - In **cache**
    - + Simple design, no need for separate buffers
    - Can evict useful demand data → cache pollution
  - In a separate **prefetch buffer**
    - + Demand data protected from prefetches → no cache pollution
    - More complex memory system design
      - Where to place the prefetch buffer
      - When to access prefetch buffer (parallel vs. serial with cache)
      - When to move the data from the prefetch buffer to cache
      - How to size the prefetch buffer
      - Keeping the prefetch buffer coherent
- Many modern systems place prefetched data into the cache
  - Many Intel, AMD, IBM systems and more ...





# Challenges in Prefetching: Where (II)

---

- Which level of cache to prefetch into?
  - Memory to L4/L3/L2, memory to L1.  
Advantages/disadvantages?
  - L3 to L2? L2 to L1? (a separate prefetcher between levels)
- Where to place the prefetched data in the cache?
  - Do we treat prefetched blocks the same as demand-fetched blocks?
  - Prefetched blocks are not known to be needed
    - With LRU, a demand block is placed into the MRU position
- Do we skew the replacement policy such that it favors the demand-fetched blocks?
  - E.g., place all prefetches into the LRU position in a way?



# Challenges in Prefetching: Where (III)

---

- **Where** to place the hardware prefetcher in the memory hierarchy?
  - In other words, what access patterns does the prefetcher see?
  - L1 hits and misses
  - L1 misses only
  - L2 misses only
- Seeing a more complete access pattern:
  - + Potentially better **accuracy** and **coverage** in prefetching
  - Prefetcher needs to examine more requests (bandwidth intensive, more ports into the prefetcher?)



# Challenges in Prefetching: How

---

- **Software** prefetching
  - ISA provides prefetch instructions
  - Programmer or compiler inserts prefetch instructions  
Usually works well only for “regular access patterns”
  - E.g., `_mm_prefetch` call
- **Hardware** prefetching
  - Specialized hardware monitors memory accesses
  - Memorizes, finds, learns address strides/patterns/correlations
  - Generates prefetch addresses automatically
- **Execution-based** prefetchers
  - A “thread” is executed to prefetch data for the main program
  - Can be generated by either software/programmer or hardware



# Cache Coherence Problem

- Suppose two CPU cores share a physical address space
  - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1



# Coherence Defined

---

- Informally: Reads return most recently written value
- Formally:
  - P writes X; P reads X (no intervening writes)  
⇒ read returns written value
  - $P_1$  writes X;  $P_2$  reads X (sufficiently later)  
⇒ read returns written value
    - c.f. CPU B reading X after step 3 in example
  - $P_1$  writes X,  $P_2$  writes X  
⇒ all processors see writes in the same order
    - End up with the same final value for X



# Cache Coherence Protocols

---

- Operations performed by caches in multiprocessors to ensure coherence
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access
- Snooping protocols
  - Each cache monitors bus reads/writes
- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory



# Invalidating Snooping Protocols

- Cache gets exclusive access to a block when it is to be written
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		0
CPU B read X	Cache miss for X	1	1	1



# MESI State-transition Diagram

---

- Definitions
  - Modified (M)
    - Cache line is present **only** in current cache
      - It is *dirty*
      - It has been modified (M state) from value in MM
  - Exclusive (E)
    - Cache line is present **only** in the current cache
    - It is *clean*
    - It matches main memory





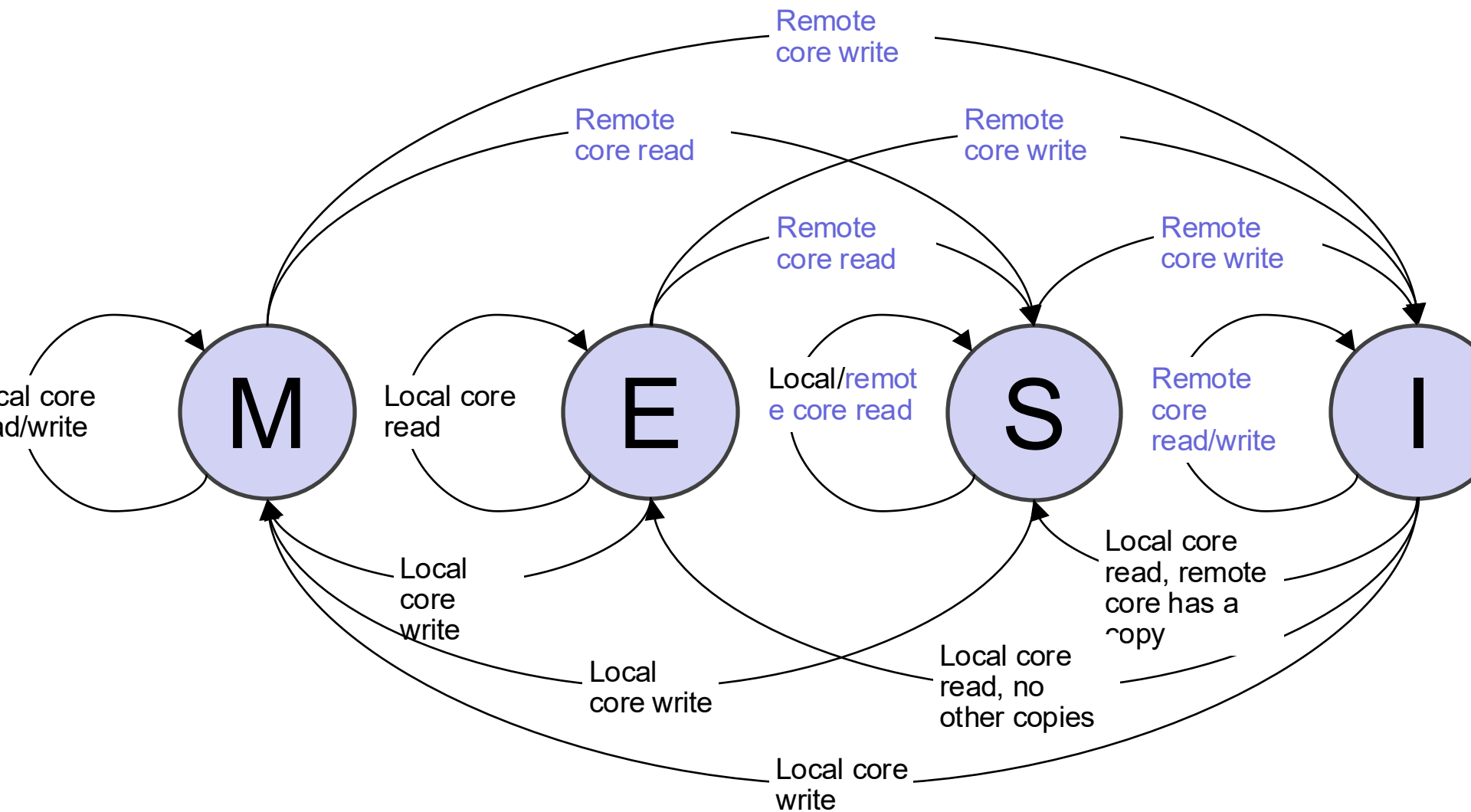
# MESI State-transition Diagram

---

- Definitions
  - Shared (S)
    - Cache line **may be stored in other caches** of the machine
    - It is *clean*
    - It matches main memory
  - Invalid (I)
    - Indicates that this cache line is invalid



# MESI State-transition Diagram





**Thanks for Your Attention!**

