Cache and Cache Consistency and Shared Memory

Hui Chen [a]

[a]CUNY Brooklyn College

November 26, 2025

Why Cache: Memory Bottleneck

Performance of computers is often limited by memory bandwidth and latency

- memory latency: time to access a single word from memory
  - processor cycle time: time for a single CPU cycle
  - access latency $\gg$ processor cycle time
- bandwidth: number of words accessed per unit of time

Why Cache: Distance and Physical Size Matters

Speed of light: $\tilde{3}0$ cm per nanosecond. Signals in wires travel at about $2/3$ speed of light

Large memory:

- Signals have further to travel
- Fan out to more circuits and locations

Memory Hierarchy

CPU – Cache (L1, L2, L3) – Main Memory – Disk Storage – Remote Storage

- smaller, faster memories closer to the CPU
- larger, slower memories further from the CPU
- data is moved between levels of the hierarchy

Characteristics:

- Capacity: increases down the hierarchy
- Access time: increases up the hierarchy
- Cost per bit: increases up the hierarchy
- Volatility: decreases down the hierarchy
- Bandwidth: on chip $\gg$ off chip

Cache Basics

Cache: smaller, faster memory that stores copies of data from frequently used slower memory locations

- temporal locality: recently accessed data likely to be accessed again soon
- spatial locality: data near recently accessed data likely to be accessed soon

Cache operation:

- on a memory access, check if the data is in cache (cache hit)
- if not (cache miss), fetch from main memory and store in cache

Cache Basics: CPU Cache

CPU – Cache (on-CPU-chip) – Main Memory (off-CPU-chip)

Address: tag index block offset

Tag	Data Block								
111...001									← Cache Line
...									

Cache Basics: CPU Cache Example for Cache Benefits

```
double a[8] = {1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0};
double b[8] = {8.0,7.0,6.0,5.0,4.0,3.0,2.0,1.0};
double sumofproducts;
for (int i = 0; i < 8; i++) {
    sumofproducts += a[i] * b[i];
}
```

How does cache help here? Assume cache line size is 64 bytes (8 doubles) and there are multiple lines and the cache is initially empty.

Cache Basics: Cache Read

- CPU issues read for address A
- Cache checks if block containing A is in cache (check tag)
- If found in cache (hit),
  - return data from cache to CPU
- If not found in cache (miss),
  - fetch *block* from main memory to cache,
  - then return data to CPU,
  - then select line and update cache (invoking replacement algorithm to select a line if full)

Cache Basics: Cache Write: Example

- CPU issues write for address A
- Cache checks if block containing A is in cache (check tag)
- If found in cache (hit),
  - update data in cache,
  - mark line as dirty (modified)
- If not found in cache (miss),
  - fetch *block* from main memory to cache,
  - then update data in cache,
  - mark line as dirty (modified),
  - then select line and update cache (invoking replacement algorithm to select a line if full)

Cache Basics: Cache Write Policies

- Cache hit
  - write through: write both cache and memory
  - write back: write cache only, memory is written only when the entry is evicted
- Cache miss
  - no write allocate: only write to main memory
  - write allocate (aka fetch on write): fetch into cache
- Common combinations
  - write through and no write allocate
  - write back with write allocate

Multiple CPUs?

- Each CPU has its own cache
- Problem: caches may have copies of the same memory location
- If one CPU updates its cache, other caches may have stale copies
- Solution: Cache Coherence Protocols

Cache Coherence: Multiple CPUs

CPU 1		CPU 2
Cache-1 7312	...	Cache-2 7312
Main Memory 7312		

Suppose CPU 1 updates its cache to 7310.

- Write back. Memory and Cache-2 have stale copies (7312).
- Write through. Memory is updated to 7310, but Cache-2 has stale copy (7312).

Cache Coherence

A cache coherence protocol ensures no updates are lost, i.e., all writes by one processor are eventually visible to other processors

Require hardware support:

- only one processor at a time has write permission for a location
- no processor can load a stale copy of the location after a write

Coherent Memory System

A memory system is coherent if:

- P.write(Location, Value), Y :=P.read(Location); Y=Value (write followed by read by same processor)
- P2.write(Location, Value), Y :=P1.read(Location); Y=Value (write by one processor followed by read by another processor)
  - Given the read and write are sufficiently separated in time
  - No other writes to Location occur between the two accesses
  - i.e., coherence provides per-location sequential consistency (two writes to the same location by any two processors are seen in the same order by all processors)

Example Designs

- Snoopy Cache
- Directory-based

Snoopy Cache

Hardware design:

- Cache controllers in CPUs work together to maintain cache coherence.
- Cache controllers send commands to the bus.
- Each cache controller snoops on the bus traffic to catch various commands and follow them.

Coherence protocols:

- MSI (Modified, Shared, Invalid)
- MESI (Modified, Exclusive, Shared, Invalid)
- MOESI (Modified, Owner, Exclusive, Shared, Invalid)

Snoopy Cache: Example Protocol - MSI

Each cache line has a state:

Tag	State	Data Blocks	

- M – Modified: cache line is valid only in current cache and has been modified (dirty)
- S – Shared: cache line may be stored in other caches and matches main memory (clean)
- I – Invalid: cache line is invalid

Semantics:

- If a cache line is in state S, then only read is possible.
- If a cache line is in state M, then write is possible as well.
- Writing to a cache line
  - If it's in state M, the cache controller does the write.
  - If it is not in state M, involving two cache controllers:
    - it sends an invalidation request to other caches, switches the state to M, and does the write;
    - other cache controllers switch the state to I.
- Reading a memory address
  - If it's a cache hit, read it.

MSI State Transition Diagram

The correctness of the MSI protocol can be verified using a state transition diagram.

https://en.wikipedia.org/wiki/MSI_protocol

Directory-Based Cache Coherence

Hardware design:

- Each memory block has a directory entry that keeps track of which caches have copies of the block.
- When a CPU wants to read or write a block, it sends a request to the directory.
- The directory coordinates the requests and ensures coherence.

Advantages:

- Scales better for large numbers of processors.
- Reduces bus traffic compared to snoopy protocols.

Summary

- Cache basics – analyze cache hits and misses
- cache coherence
  - snoopy cache coherence
  - directory-based cache coherence

Why Shared Memory?

Distributed applications need to share data among multiple processes on multiple hosts.

- Message passing: processes communicate by sending and receiving messages
  - often no synchronization needed (locks) between sender and receiver
  - often used when there are multiple writers
- Shared memory: processes communicate by reading and writing to a shared memory space
  - typically require explicit synchronization (locks) between readers and writers
  - often used when there are multiple readers, but no writers (read-only)

Distributed Shared Memory (DSM)

DSM for processes: different processes running on different hosts sharing a memory page.

- shared memory page is mapped into the address space of each process
- processes read and write to the shared memory page as if it were local
- underlying DSM system handles communication and synchronization

DSM Synchronization Strategies

- write-update
- write-invalidate

DSM Synchronization Strategies: Write-Update

- When a process updates a memory page, the update is multicast to all other replicas.
- The multicast protocol determines consistency guarantees (e.g., FIFO-total for sequential consistency).
  - All processes see the same sequence of updates to the shared memory page.
- Reads are cheap (always local), but writes are costly (always multicast).

DSM Synchronization Strategies: Write-Invalidate

- two states for a shared page, i.e., read-only or read & write
- Read-only: the memory page is potentially replicated on two or more processes/machines
- Read & write: the memory page is exclusive for the process (no other replica)
- If a process intends to write to a read-only page, an invalidate request is multicast to other processes; later writes can take place without communication (cheap).
- Writes are only propagated when there's a read by another process (cheap for write, costly for read). But a write can be delayed by invalidation (costly for write).

Granularity Problem

Assume the unit of share is a page:

- When two processes on two hosts share a page, it doesn't always mean that they share everything on the page.
- P1@Host1.read(X), P2@Host2.read(X), P1@Host1.read(Y), P2@Host2.read(Z), and X, Y, and Z are on the same page.

Granularity Problem

True sharing

- Two processes share the exact same data.

False sharing

- Two processes do not share the exact same data, but they access different data from the same page.

Problems with False Sharing

- Write-invalidate: unnecessary invalidations
- Write-update: unnecessary data transfers

Granularity Problem: Solutions

Change unit of sharing (change page size)

- Increase page sizes
  - Better handling for updates of large amounts of data (good)
  - Less management overhead due to a smaller number of units/pages to handle (good)
  - More possibility for false sharing (bad)
- Reduce page sizes
  - The opposite of the above
  - If there is an update of a large amount of data, it'll be broken down to many small updates, which leads to more network overhead (bad)
  - A smaller page size means more pages, which leads to more management overhead, i.e., more tracking of reads and writes (bad)
  - Less possibility of false sharing (good)

Summary

- Why shared memory?
- Distributed shared memory (DSM)
- DSM synchronization strategies
- Granularity problem