Computer Architecture Memory hierarchy

http://d3s.mff.cuni.cz/teaching/nswi143



Lubomír Bulej

bulej@d3s.mff.cuni.cz

CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

The memory wall

CPU performance limited by memory performance

- CPU performance grows faster than memory performance
 - Simple operations take tenths of ns
 - Access to memory takes tens of ns
 - Pick two of three: memory as fast as the CPU, sufficient capacity, reasonable price



The memory wall (2)

- Burks, Goldstine, von Neumann: Preliminary discussion of the logical design of an electronic computing instrument (1946)
 - Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available [...] We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible."



The memory wall (3)

Analogy: books and library

- Library
 - Lots of books, slow access (walk to the library)
 - Library size (finding the right book takes some time)
- How to avoid high latency?
 - Borrow some of the books
 - Leave them on a desk or a shelf
 - Often-used books at hand (temporal locality)
 - Borrow more books on the same topic (spatial locality)
 - Think of what will be needed next (pre-fetching)
 - Both the desk and the self have limited capacity



The memory wall (4)

• How to get over the memory wall?

- Exploit locality of memory accesses
 - Property of most real (useful) programs
 - Applies to both instructions and data

Temporal locality

■ Recently used data are likely to be accessed again in near future → keep such data in a small, but very fast memory

Spatial locality

■ Data near recently used data are likely to be accessed in near future → access data in larger blocks (that include data in close proximity)



Volatile memory

• Static RAM

- Primary goal: speed
- Secondary goal: capacity
 - 6 transistors per 1 bit, speed depends on area (latency for small capacities can be < 1 ns)
- Combines well with other CPU logic
- Contents stay in memory as long as it is powered
 - No need for periodic refresh



Static RAM









Static RAM cell

• Pair of inverters + control transistors

6 transistors per 1-bit cell



Static RAM in matrix arrangement

• M × N bits: M rows of N bits

bl[1] bl[2] bl[0] Row selection Binary to 1-hot Access in two steps wl[0] 1. Row selection (word lines) 2. Column read wl[1] (bit lines) wl[2]

Static RAM (2)



Volatile memory (2)

Dynamic RAM

- Primary goal: density (cost per bit)
 - 1 transistor + 1 capacitor per 1 bit
 - High latency
 - 40 ns internally
 - 100 ns between circuits
- Contents deteriorate over time
 - Needs periodic refresh (read data and write it back)
- Difficult to combine with CPU logic
 - Different manufacturing process



Dynamic RAM cell

Capacitor + control transistor

Information stored as electric charge bl[0] bl[1] Capacitor charges/discharges itself due to losses and content of surrounding cells Reading is destructive (value read is immediately written back) bit line wl[0] wl[1] word line sense amplifiers

Dynamic RAM



Increasing DRAM performance

Observation

- Reading DRAM row takes the most time
- Row contains more than just the requested word
- Need to amortize the cost of reading a row
 - Use more words from last read row
 - Pipelining data output and selecting (reading) new row
 - The recently read row is stored in an output register, start reading another row while data is being transmitted from the data register to the CPU (via bus or other interconnects)



Exploiting locality of access



Hierarchy of memory components



Hierarchy of memory components (2)

Back to the library analogy

- - Only one page
- Primary (level 1) cache \leftrightarrow books on a desk
 - Actively used, very quick access, small desk capacity
- Secondary (level 2) cache \leftrightarrow books on a shelf
 - Actively used, relatively quick access, medium capacity
- Main memory \leftrightarrow library
 - Almost all data, slow access, huge capacity
- Swap memory ↔ inter-library borrowing
 - Very slow, very rare



17/73

Cache

Big and fast memory (an illusion)

- Data transfer between cache levels handled by hardware
 - Automatically finds missing data
 - Cache controller
 - Integrated on-chip SRAM
 - Software can provide hints
- Cache organization (ABC)
 - Associativity, Block size, Capacity
 - 3C model of cache misses



Mapping memory directly to cache

Basic structure

- An array of cache lines
 - E.g., 1024 cache lines, 64 B each \rightarrow 64 KB
- "Hardware hash table" based on address
 - Example configuration for 32bit addresses
 - 64-byte blocks → the lowest 6 bits addresses the byte in a block (offset bits)
 - 1024 blocks → next 10 bits represents block number (index bits)
 - What about the remaining 16 bits?



Mapping memory directly to cache (2)



Mapping memory directly to cache (3)

Finding the correct data

- Each cache line can contain one of 2¹⁶ different blocks of main memory with identical cache line number (in our example)
 - "Hash function"
 - Detect the matching data
 - Flag indicating cache line validity (valid bit)
 - Tag with the remaining bits of the address (tag bits)

Algorithm

- 1. Read cache line determined by the index
- 2. If the *valid bit* is set and the *tag* matches the bits in the rest of the address, then it is a *cache hit*
- 3. Otherwise *cache miss*



Mapping memory directly to cache (4)



Metadata overhead (tags, valid bits)

• 64 KB cache with 1024 cache lines of 64B each

- For 32-bit addresses
 - (16 bits per tag + 1 valid bit) × 1024 cache lines ~ 2,1 KB
 - Overhead 3,3 %
- For 64-bit addresses
 - (48 bits per tag + 1 valid bit) × 1024 cache lines ~ 6,1 KB
 - Overhead 9,6 %



Handling a cache miss

Read data into cache

- Cache controller
 - Sequential circuit/state machine
 - Requests data from the next level of memory hierarchy using the memory address that caused the cache miss
 - Writes data, tag, and valid bit into cache line
- Cache misses cause pipeline stalls
 - Stall logic controlled by the cache miss signal
 - Scalar pipeline stalls (situation similar to data hazard)
 - Superscalar pipeline can still execute other instructions
 - Cache operates independently from the data path
 - CPU supports multiple in-flight memory operations



Cache performance

Cache operations

- Cache access (load/store from/to memory)
- Cache hit (data found in cache and passed to data path)
- Cache miss (data not found, trigger load from next level)
- Cache fill (loading data into cache)

Metrics characterizing cache performance

- %miss: fraction of cache misses from all cache accesses (miss rate)
- *t_{hit}*: time needed to access cache (and to get data on cache hit)
- *t_{miss}*: time needed to read data into cache

Goal: minimize average access time

tavg = thit + tmiss × %miss



Reducing miss rate

• Obvious approach: increase capacity

- Miss rate decreases monotonically
 - Law of diminishing returns
- thit increases with square root of capacity

Less obvious approach

- Change cache organization
 - Capacity is restricted by transistor budget
 - Try to get the most out of available resources



%_{miss} "working set" size [1] Cache Capacity

Cache organization: block size

Increasing block (cache line) size

- Attempt to exploit spatial locality
- Decrease the number of cache lines
 - Moves the split between index and offset bits of an address
 - The number of tag bits does not change

Consequences

- Reduces miss rate (to a degree)
- Less tag overhead (smaller number of cache lines)
- More potentially useless data transfers
- Premature replacement of potentially useful data



Block (cache line) size vs miss rate

Spatial prefetching

Changes miss/miss to miss/hit for blocks with consecutive addresses (stored in consecutive blocks)

• Interference

- Changes hit to miss for blocks with non-consecutive addresses stored in consecutive cache blocks
 - Prevents having both blocks in the cache at the same time

Always: mix of both effects

- Spatial prefetching dominates initially, interference kicks in later
 - Limit case: single (huge) cache line
- Common (block) cache line size: 16 128 B

[1]

Block Size

Block (cache line) size vs fill latency

• In principle

Reading long cache lines should take longer

• In practice

- *t_{miss}* does not change too much for isolated misses
 - Critical Word First / Early Restart
 - Cache controller and memory cooperate to first transmit the word actually requested by the CPU (to minimize pipeline delays)
 - The rest of the cache line contents follow
- *t_{miss}* increases when misses occur in batches
 - Cannot read/transmit/fill multiple lines at the same time
 - Limited bandwidth between memory and the CPU
 - Delays of queued requests accumulate



Caches with associative mapping

Set associativity

- Goal: reduce conflicts between memory blocks mapped to the same cache line
 - A block of memory can be stored in any block from a set of cache lines
- Groups of blocks (cache lines) = sets
- Cache line in a set = way
 - Example: 2-way set-associative cache
 - Limit cases
 - 1 way: directly mapped cache
 - 1 set: fully-associative cache
- Increases t_{hit}
 - Selecting data from cache lines in the set



Caches with associative mapping (2)



Caches with associative mapping (3)

Looking for data

- 1. Use index bits of the address to select a candidate set of cache lines where to look for data
- 2. In parallel: fetch data and tags from all ways
- 3. In parallel: compare all tags with address tag bits
- Impact on tag/index bit split (constant capacity)
 - More ways \rightarrow less sets \rightarrow less index bits
 - More tag bits



Cache associativity vs miss rate

Increasing associativity

- Reduces miss rate
 - Law of diminishing returns
- Increases t_{hit}



- Note: *n*-way associative cache where *n* is not a power of 2 is feasible
 - Not commonly found though
 - Cache line (block) size should be power of 2
 - Number of sets should be power of 2
 - Simplifies addressing (simply cut off address bits)



Fully associative cache

- Set-associative with 1 set (number of ways = number of blocks)
 - A block of memory can be in any block of cache (cache line)
 - All bits of address bits (except offset bits) represent a tag
 - Associative memory
 - Contents addressed by a key (key = tag)
 - Analogy: key/value pairs in a hash map



Fully associative cache (2)



3C model: understanding cache misses

Cache miss classification

Compulsory (cold) miss

- "Ain't seen no such address before"
- Cache miss would occur even in infinitely large cache

Capacity miss

- Caused by insufficient cache capacity
 - Repeated access to a block of memory separated by at least N accesses to N other blocks (where N is number of blocks in cache)
 - The working set is simply too large
- Cache miss would occur even in fully-associative cache

Conflict miss

- All the (remaining) misses that are not cold/capacity
- Caused by low cache associativity


Miss rate: ABC

Consequences of the 3C model

- Increasing associativity does not help if there are no conflict misses
- Associativity
 - Reduces the number of conflict misses
 - Increases thit
- Block (cache line) size
 - Increases the number of conflict/capacity misses (less cache lines)
 - Reduces number of cold/capacity misses (spatial locality)
 - Does not (mostly) influence thit
- **C**apacity
 - Reduces the number of capacity misses
 - Increases thit



Reading data from cache

Tag and (all) data can be read in parallel

- If a tag does not match, data is not used (cache miss)
 - Issues a fill request to lower layer of the memory hierarchy

Read miss: where to put data from lower layer?

- Replace contents of "some" cache line with new data
 - The original content is evicted
- Direct-mapped cache
 - The cache line to evict is determined by address index bits
- (Set) associative cache
 - All ways within a set are potential candidates, need to pick a "victim"
 - Ideally: don't throw away data that will be needed soon
 - Random
 - LRU (Least Recently Used): optimal wrt. temporal locality
 - NMRU (Not Most Recently Used): approximates LRU



Writing data to cache

Write hit

- Data can be written to the cache line
 - If writing to cache only, the data in cache and main memory will be inconsistent
- Write through
 - Data always stored both to cache AND lower layer/memory on store operation
 - Problem: memory operations (a thus store instructions) take too long
 - Solution: write data to cache and a write buffer (output towards lower layer)
 - CPU only needs to wait if the write buffer is full (How can that happen?)
 - Write buffer needs to be checked when looking for data in cache
- Write-back
 - Data only stored to cache on store operation
 - Requires a "dirty bit" to indicate if a cache line has been modified wrt. lower layer
 - Modified (dirty) cache line only written to memory when evicted
 - Improves performance when a program issues stores as fast or faster than the memory can handle
 - More difficult to implement



Writing data to cache (2)

Write miss in a write-through cache

- Reading tag/writing data can be done at the same time
 - Overwriting wrong data is not a big problem, the right data is in memory
- Write allocate
 - Fill cache with data from lower layer first
 - Continue as if it was write hit: replace part of cache line with new data
 - Can prevent cache misses on future accesses (locality)
 - Not always the case
 - Requires sufficient memory bandwidth
- Write no-allocate
 - Data written only to the lower layer
 - Eliminates fill from lower layer on write miss
 - Suitable for "pass though" data that the CPU only touches once
 - Zeroing a page, writing block of data to disk, sending a packet over network
- Processors often allow setting a write strategy for individual pages



Writing data to cache (3)

Write miss in a write-back cache

- Cannot always read tag and write data at the same time
 - Modified cache line to be evicted first needs to be written to lower level of the hierarchy
- Write requires two steps...
 - Check for hit/miss and then write
- ... or using a store buffer (input from data path)
 - Check hit/miss while putting data into buffer
 - Data written from store buffer to cache on write hit
- Writing into a modified cache line (on replacement)
 - Data moved from cache line to write-back/victim buffer, stored to lower layer/memory later (asynchronously)
 - When looking for data in cache, the write-back buffer needs to be consulted too



Multi-level caches (1)

• Goal: reduce cache miss penalty

- 1-level cache: Total CPI = 1.0 + Memory stall cycles per instruction
 - 4 GHz processor, memory access 100 ns (400 cycles), 2% cache misses: Total CPI = 1.0 + 2% x 400 = 9
- 2-level cache: Total CPI = 1.0 + Primary stalls per instruction + Secondary stalls per instruction
 - Hit/miss latency 5 ns (20 cycles), reduces miss rate to 0.5%: Total CPI = 1.0 + 2% x 20 + 0.5% x 400 = 3.4



Multi-level caches (2)

Different cache levels have different roles

Allows optimizing for different criteria than with singlelevel cache

Primary cache (L1)

- Minimize hit time
- Allows increasing clock rate or reducing number of pipeline stages
- Typically smaller capacity, smaller cache lines (lower penalty for cache miss)

Secondary cache (L2)

- Minimize miss rate
- Reduces penalty for accessing memory
- Significantly larger capacity (access time not critical), larger cache lines, higher associativity (focus on reducing miss rate)



Do we need to know (about caches)?

• Quick Sort vs. Radix Sort

- LaMarca, Ladner (1996)
- O(n×log n) vs. O(n)
- Theory: nothing of interest





Do we need to know (about caches)? (2)

• Surprise: Quick Sort turned out to be faster for larger amounts of data ...



Do we need to know (about caches)? (3)

Theory vs practice

The way the Radix Sort implementation accessed data caused too many cache misses



Do we need to know (about caches)? (4)

Solution

Modify Radix Sort implementation to first process data in a block of memory that is already in cache (cache line)



Memory dominates CPU performance

The memory wall

- CPU performance grows faster than memory performance
- There is no ideal memory technology
 - Fast, large, cheap pick two out of three
- Locality in accessing memory
 - Temporal + spatial, found in real (useful) programs
- Solution: hierarchy of memories
 - Optimize average time to access memory
 - Different technologies in different levels
 - Transfer data between levels

Cache: an illusion of ideal memory

- 1-3 levels of fast memory between CPU and main memory
 - SRAM, L1 capacity ~ 64KiB, L2/L3 capacity ~ 256KiB-16MiB
 - Transparent from programmer's (and CPU) perspective
 - CPU (data path) only requests data from cache
 - Data transfers between cache and memory handled by HW
 - Data stored in blocks (cache lines) corresponding to blocks of memory
 - tag part of an address that disambiguates the mapping of memory blocks to cache blocks
- 9 3C model: cache miss classification
 - Changing cache organization to eliminate cache misses
- ABC: basic cache parameters
 - Associativity, block (cache line) size, capacity



Memory hierarchy and parallelism

Multiprocessor systems with shared memory

- Programs read and write to shared variables
 - Processors issue read/write requests for specific memory addresses
- Intuitive expectation
 - Reading from a memory address always produces the last value written to that address by any other processor





The problem with caches...

• Contents of memory replicated in local caches

- A necessity in modern processors
- The (local) cache of different processors could potentially contain different value for the same memory location





The problem with caches... (2)

Also exists in single-processor systems

- DMA transfers between IO devices and memory
- Both the device and the processor could read stale data

Usual solution

- Buffer memory stored in an uncached physical page
- Force cache flush after completing work on buffer contents





The cache coherence problem

- Reading from memory address X should return the last value written to address X by any processor
 - Intuitive expectation for a shared memory system
 - Problem with multiple clients
 - Single-processor system with DMA transfers
 - Multi-processor systems

• Cause: presence of local and global state

- Multiple local states in processor caches
- Global state in main memory



Problems with the intuitive expectation

• Not well defined for a multi-processor system

Reading from memory address always returns the last value written to that address by any processor

• What does "the last" mean?

- What if two processors write at the same time?
- What if a write by P1 followed by a read from P2 are so close (in time) that it's impossible to communicate the occurrence of the read to other processors?

 In sequential programs, "last" is determined by program order (not time)

- Also holds within a thread of a parallel program.
- Inadequate for multiple parallel threads.



Coherent shared memory system

Processor observes its *own writes* in *program order*

- Intuitive requirement for single-processor systems
- A read from address X by processor P that follows a write to address X by processor P returns the value written by P
 - Assuming no other processor wrote to X in between
- Writes to memory eventually observed by all processors
 - A read from address X by processor P that follows a write to address X by processor Q returns the value written by Q if the read and write are *sufficiently separated in time*
 - Assuming no other processor wrote to X in between
 - Does not define when the news of a write is propagated.
- Writes to the same location are serialized (ordered)
 - Two writes to the same location by any two processors are observed in the same order by all processors.



Write serialization (ordering)

Writes to the same location are serialized (ordered)

Two writes to the same location by any two processors must be observed in the same order by all processors.

• Example

- P1 writes value **a** to **X**. Then P2 writes value **b** to **X**.
- Consider the following situation
 - P1 first observes write of **a** to **X** and then the write of **b** to **X**
 - P2 first observes write of **b** do **X** and then the write of **a** to **X**
- There is no global ordering of loads and stores to X that would produce the results of such a parallel program
 - For a coherent system, a global ordering must exist that is consistent with the result of program execution



Coherence vs. consistency

• Coherence

Defines read/write behavior with respect to the same memory location

Consistency

- Defines read/write behavior with respect to different locations
- Deals with the when of write propagation

• For our purposes

If a processor writes to address X and then to address Y, then any processor that observes the result of the write to Y also observes the result of write to X.



Implementing coherence

• Hardware solutions

- HW ensures that a read from a memory location by any processor returns the last value written to that place
 - For a suitable/reasonable definition of "last"
- Metadata keeps information about data in cache with respect to others
- Solutions: invalidating or updating data in cache
 - Coherence protocol: rules for updating the metadata of a particular cache line (block) in the whole system



Simple solution: shared cache

Issues with scaling

Interference, contention

Potential advantages

- Fine sharing granularity (overlap of working sets)
- Actions by one processor may prefetch data for others





Solutions based on snooping

Processors (cache controllers) on shared interconnect

All events related to coherence **broadcast** to all processors (cache controllers) in the system.

Cache controllers snoop on (monitor) all memory operations

- Each (individually) reacting so as to ensure coherency
- Must react to events from both the processor side (data path) and the interconnect side (activity of other processors)





Simple coherence implementation

• Write-through cache

- Coherence granularity: one cache line
- Invalidates a cache line in other processors on write
 - Broadcast write to a cache line on the shared interconnect
 - Other processors invalidate the cache line in their local caches
 - Next read of that cache line by other processors will result in a cache miss
 - The other processors are forced to retrieve the updated value from memory





Basic Valid/Invalid (VI) protocol



Write-through policy is inefficient

- Every write propagated to memory
 - Very high bandwidth requirements
- Write-back cache absorb most writes
 - Significantly reduces bandwidth requirements
 - How to ensure write propagation/serialization?
 - Requires a more sophisticated protocol



Invalidation protocol for write-back cache

Cache line in exclusive state can be modified without notifying the other caches

Other caches don't have the cache line, therefore other processors cannot read data from it without issuing a memory read request

• Writes only possible to cache lines in exclusive state

- If a processors wants to write to a cache line that is not in exclusive state, the cache controller first broadcasts a readexclusive transaction
 - Tells the other caches about an impending write
- Note: read-exclusive broadcast is also required for cache lines already present in cache (to upgrade their state)
- Dirty cache line is necessarily exclusive
- When cache controller snoops a read-exclusive request...
 - If it concerns a cache line it contains, it must invalidate i



Basic MSI invalidation protocol



Key tasks

- Obtain exclusive access for write
- Locate the most recent value on cache miss

Protocol states

- I: invalid cache line
- S: clean cache line in one or more caches
- M: dirty cache line in exactly one cache (dirty or exclusive state)

Protocol actions

- Processor Read (PrRd)
- Processor Write (PrWr)
- Bus Read (BusRd)
 - Read cache line with no intent to modify
- Bus Read Exclusive (BusRdX)
 - Read cache line with intent to modify
- Bus Write Back (flush)
 - Write cache line out to memomry



Does MSI satisfy coherence requirements?

Write propagation

Through invalidation.

Write ordering

- Writes that appear on the bus are ordered by the order they appear on the bus (BusRdX)
- Reads that appear on the bus are ordered by the order they appear on the bus (BusRd)
- Writes that don't appear on the bus (cache line already in M state)
 - Sequence of writes to line comes between two bus transactions for the line.
 - All writes in sequence performed by the same processor P (that observes them in the correct order)
 - All other processors will observe the writes to the cache line after a bus transaction for the line. All the writes will come before the transaction.
 - All processors observe writes in the same order



MESI invalidation protocol

- MSI requires 2 transactions for common case of reading and then modifying data
 - 1. transaction: BusRd to move from I state to S state.
 - 2. transaction: BusRdX to move from S state to M state.
- Even when a cache line is never shared
- Solution: new state E (exclusive clean)
 - Cache line is not modified, but exists in exactly one cache
 - Decouples exclusivity from cache line ownership (cache line not dirty, therefore copy in memory is a valid copy)
 - Upgrade from E to M does not require a bus transaction



MESI invalidation protocol (2)



More efficient (and complex) protocols

MOESI (AMD Opteron)

- Downgrade from M to S in MESI requires a memory write
- MOESI adds O state (owned, not exclusive) without memory write (cache line remains dirty)
- Other processors can have a cache line in S state, exactly one processor has a cache line in state O
- Data in memory are stale, cache controller with a cache line in O state needs to service cache misses for other processors

MESIF (Intel)

- Like MESI, but one cache keeps a shared cache line in F state (forward) instead of S
- Cache with a cache line in F state services cache miss
- Cache that was the last to read a cache line keeps the line in F state
 - The F state migrates to the last cache that read it after a read miss
 - It is assumed that the cache does not drop the cache line immediately (so that F state is not lost)
- Simplifies the decision about who services a cache miss



Consequences of implementing coherence

• Each cache must snoop and react on coherence events broadcast on shared interconnect

- Necessary to duplicate cache tags so that looking up a tag does not interfere with load/store requests from the processor
- Higher utilization of shared interconnect
 - Can be significant/limiting for high number of cores
- Some CPUs don't implement coherency at all, or only in a limited form
 - Overhead is too high, less applicable in graphic applications



Consequences for a programmer

• Is there a performance problem?

```
// per-thread event counters
int per_thread_counters [NUM_THREADS];
```

```
void thread_worker(int worker_id) {
    for (int i = 0; i < 1000; i++) {
        // ... do some work ...
        per_thread_counters[worker_id]++;
    }
}</pre>
```



Consequences for a programmer (2)

Coded with "mechanical sympathy"

```
struct PaddedCounter {
    int counter;
    char padding [CACHE_LINE_SIZE - sizeof (int)];
};
```

```
// per-thread event counters
PaddedCounter per_thread_counters [NUM_THREADS];
```

```
void thread_worker(int worker_id) {
   for (int i = 0; i < 1000; i++) {
        // ... do some work ...
        per_thread_counters[worker_id].counter++;
     }
}</pre>
```


False sharing

• Two threads write to different variables in the same cache line

- Cache line jumps between caches of writing processors
- Coherence protocol causes intensive communication between caches even though the threads don't communicate (and don't synchronize) at all
- All communication an unwanted product of false sharing
- Can significantly impact program performance on architectures implementing coherency
 - Most commonly available processors
 - Regardless of the programming language

