Memory and Caches

CPSC 424/524
Lecture #3
September 10, 2018

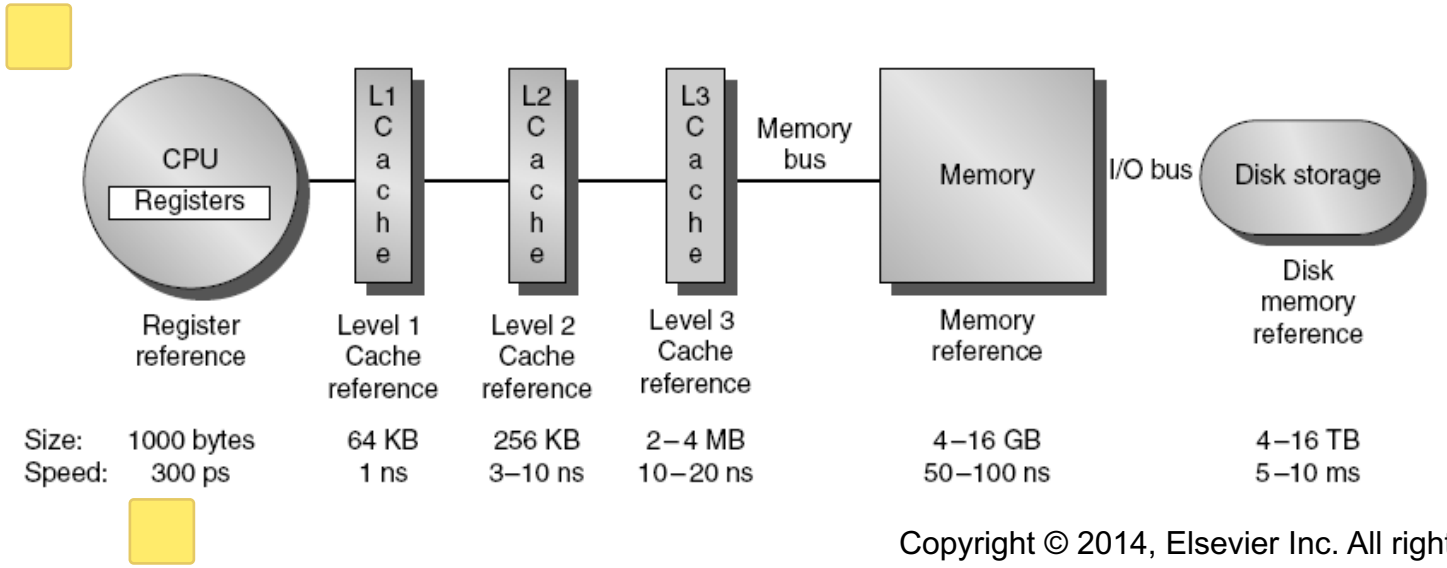
Credit: Many of these slides are based on slides from Prof. Gerhard Wellein, who developed them for use in HPC programming courses at University of Erlangen, Germany

Memory Performance

- Memory performance becomes more crucial with multi-core CPUs:
  - Aggregate peak performance requirements grow with # cores:
    - In each cycle, Nehalem (Intel Core i7) cores can generate:
      - Two 64-bit data references
      - One 128-bit instruction reference
    - With four cores and 3.2 GHz clock (for example)
      - 25.6 billion 64-bit data references/second +
      - 12.8 billion 128-bit instruction references
      - = 409.6 GB/s!
    - DRAM bandwidth is only 6% of this (25 GB/s)
    - Solution: Multi-level memory hierarchies
      - Multi-port, pipelined caches
      - Two levels of cache per core
      - Shared third-level cache on chip

Memory Hierarchies

- Memory system includes 2 or more levels of memory caches
  - Entire addressable memory space available in largest, slowest memory
  - Incrementally smaller and faster memories, each containing a subset of the memory below it, proceed in steps up toward the processor
- Temporal and spatial locality may ensure that nearly all references can be found in smaller memories
  - Gives the illusion of a large, fast memory being presented to the processor



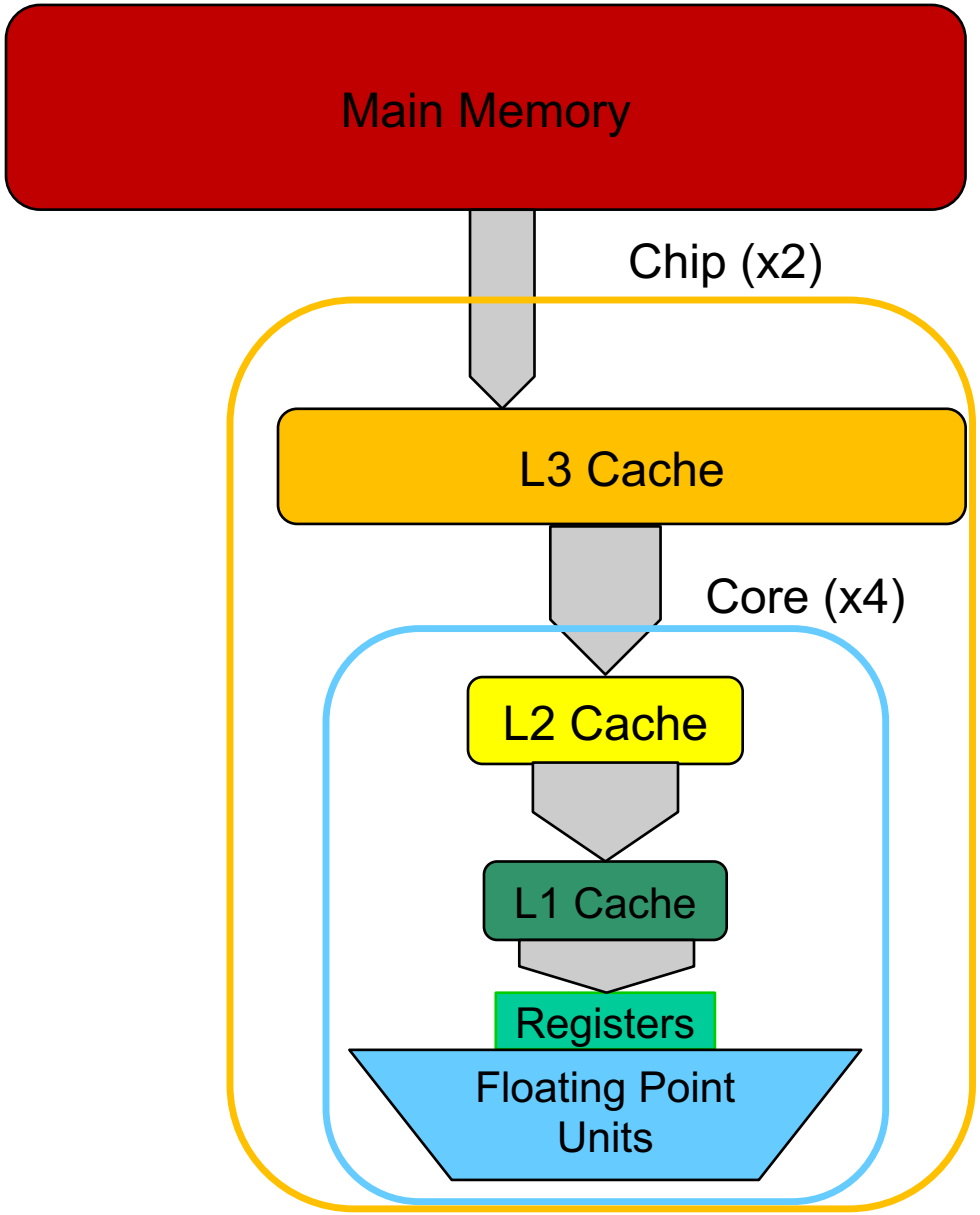
Schematic of Cache Logic

CPU/Arithmetic unit issues a LOAD request to transfer a data item to a register

Cache logic automatically checks all cache levels to see if data item is already in cache.

If data item is in cache ("cache hit") it is loaded to register.

If data item is NOT in any cache level ("cache miss") data item is loaded from main memory and a copy is held in cache.



Courtesy of Prof. G. Wellein, U. of Erlangen

Memory Latency & Bandwidth

Two quantities characterize the quality of each memory hierarchy:

- **Latency (T_{lat}): Time to set up the memory transfer from source (main memory or caches) to destination (registers).**

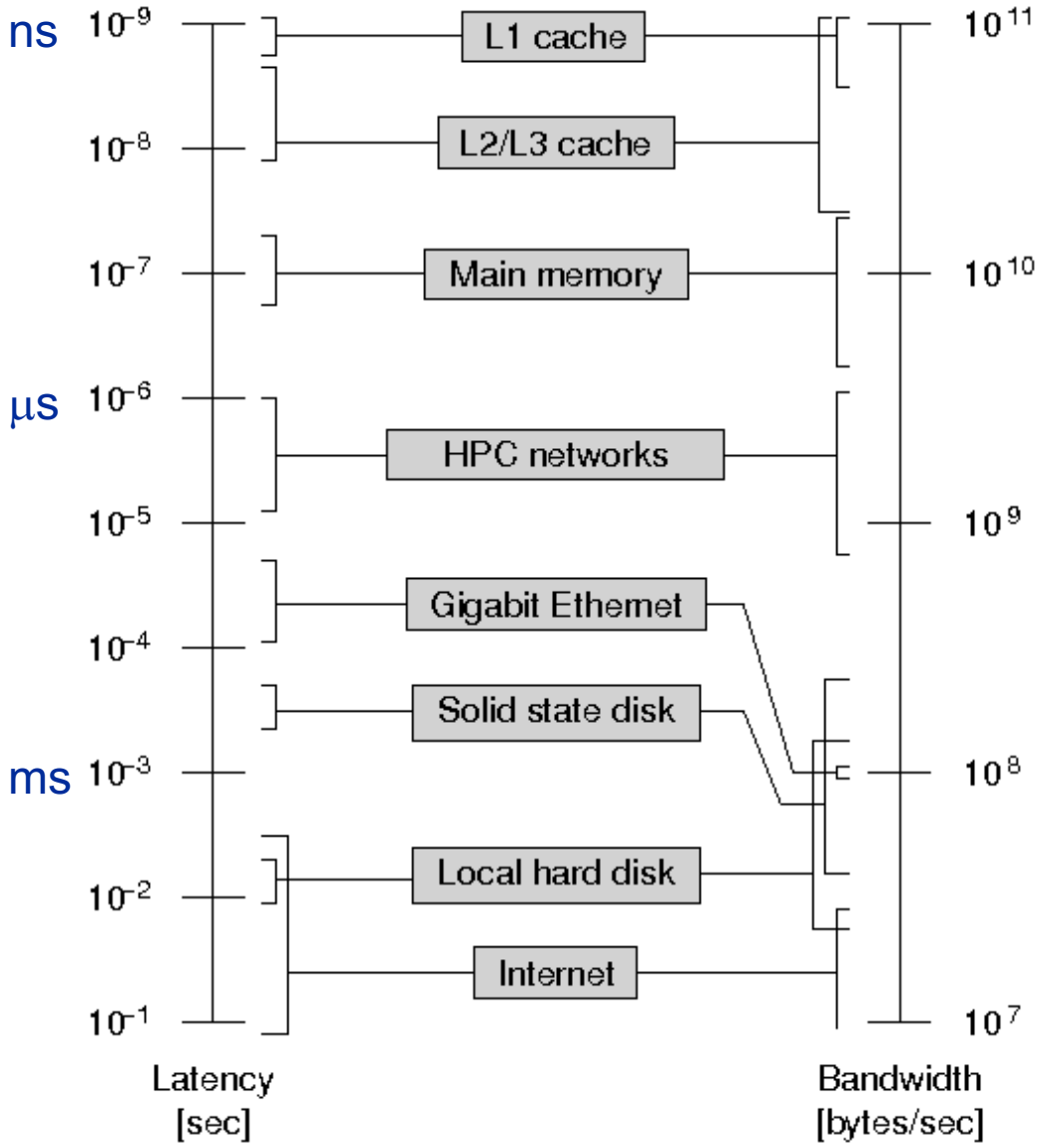
- **Bandwidth (BW): Maximum amount of data which can be transferred per second between source (main memory or caches) and destination (registers).**

- **Transfer time: $T = T_{lat} +$ (amount of data) / BW**

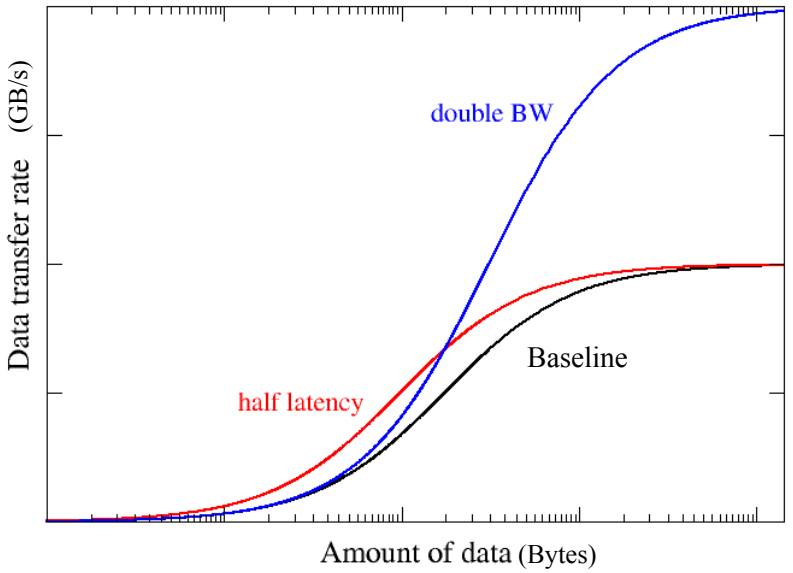
- **Data transfer rate: BW_{eff} = (amount of data) / T ("Effective bandwidth")**
  - "Small" amount of data: $BW_{eff} << BW$
  - "Large" amount of data: $BW_{eff} \sim BW$

Courtesy of Prof. G. Wellein, U. of Erlangen

Memory Latency & Bandwidth (2)



Plot below illustrates the effect of changes in the latency or bandwidth.



Courtesy of Prof. G. Wellein, U. of Erlangen

Memory Latency & Bandwidth (3)

- **Typical values for modern microprocessors:**
  - T_{lat}=100 ns; BW=4 GB/s – amount of data=8 Byte (double)
    - → T=102 ns (100 ns from latency!)
    - → Data transfer rate: 8 B / 102 ns = 0.078 B/ns = 0.078 GB/s

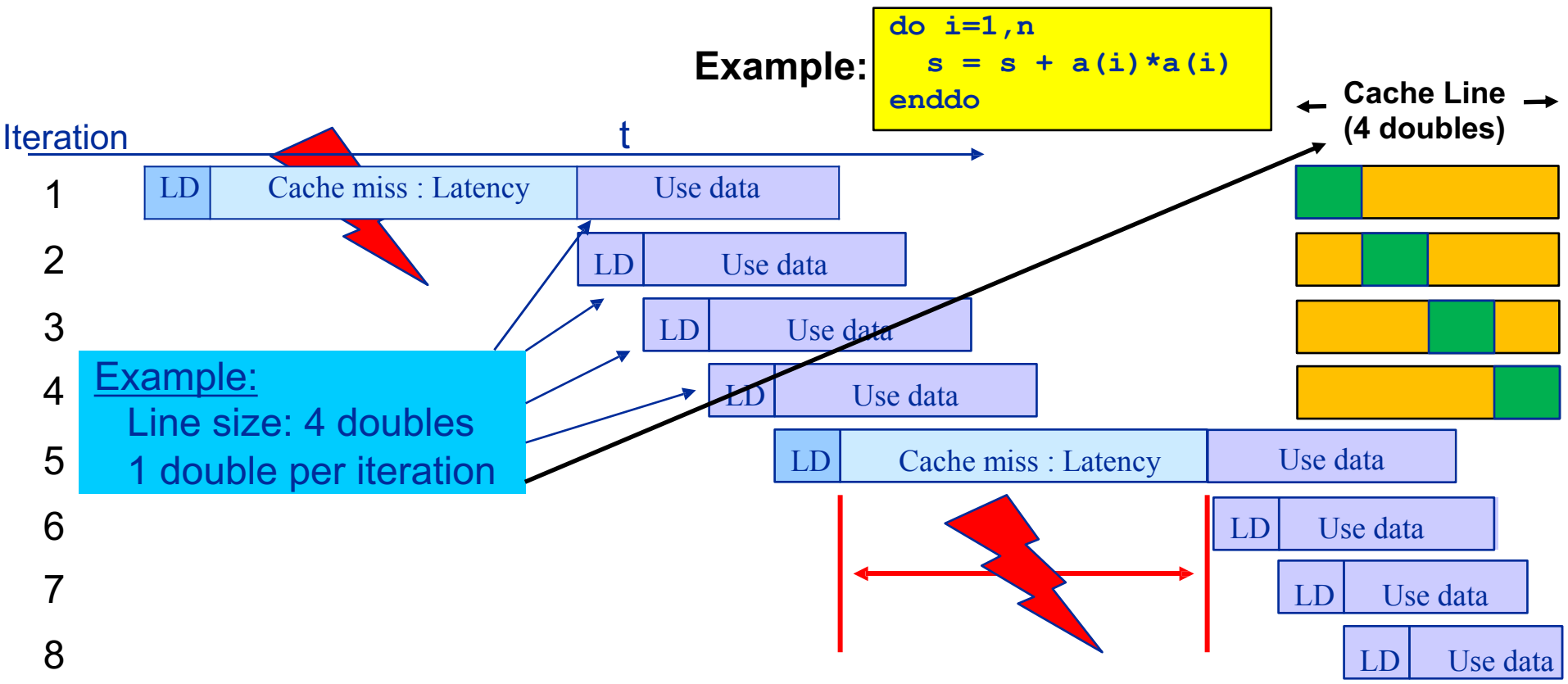
- **To improve efficiency, data access is organized in cache lines (aka "cache blocks") (CL) that are transferred as a whole**
  - e.g., if amount of data in a CL is 64 Bytes (8 doubles)
    - → T=116 ns (100 ns from latency)
    - → Data transfer rate: 64 B / 116 ns = 0.55 B/ns = 0.55 GB/s

- **Data transfers between memory and cache always happen at the cache line granularity!**

- **Still not sufficient to hide most of the memory latency**
  - Multiple non-blocking cache line transfers are supported
  - Automatic hardware prefetcher (see later)

Memory Latency & Bandwidth (4)

- When there is a cache miss for any datum, the hardware loads the whole cache line containing the datum
- Cache lines are contiguous in main memory, i.e. "neighboring" items are loaded and can then be used from cache.



Prefetching

- **CL latency is too large to fully hide it behind other operations**
- **Prefetch (PFT) instructions:**
  - Transfer one cache line from memory to cache and then issue LD to registers
  - Limited use….
- **Most architectures (Intel/AMD x86, IBM Power) use Hardware-based automatic prefetch mechanisms**
  - HW detects regular, consecutive memory access patterns (streams) and prefetches at will (subject to cache sizes and number of registers)
  - Intel x86: "Adjacent cache line prefetch" loads 2 (64-byte) cache lines on L3 miss → Effectively doubles line length on loads (typically enabled in BIOS)
  - Intel x86: Hardware prefetcher: Prefetches complete page (4 KB) if 2 successive CLs in the page are accessed
- **For streaming data access main memory latency is often not an issue!**

Principle of Locality

- Accessing one location is followed by an access of a nearby location.
- Spatial locality – accessing nearby locations consecutively.
- Temporal locality – using/reusing same or nearby locations “soon”

Spatial Locality: Exploiting Cache Lines

- **Cache line addresses latency problem – not bandwidth bottleneck**
  - Typical CL sizes: 64 Byte (on Nehalem) or 128 Byte

- **"Spatial locality": Ensures fast access to "nearby" data items**
  - Cache line use is optimal for contiguous access ("stride 1") → **STREAMING**
  - Calculations get cache bandwidth inside the cache line, but main memory bandwidth still limits the speed of the cache line transfer
  - Non-consecutive ("strided") access reduces performance
  - Access with wrong stride (e.g. cache line size) can lead to disastrous performance breakdown

GOOD ("Streaming")

```
do i=1,n
  s = s + a(i)*a(i)
enddo
```

BAD ("Strided")

```
do i=1,n,2
  s = s + a(i)*a(i)
enddo
```

a(1:n) is still loaded from main memory: same runtime, but half the flops!

→ Performance of strided loop is half of the streaming one

Dealing with Cache Size

- **If cache is full, "old data items" need to be removed when new data items come in → Cache lines wear out**
- **"Age" of cache line ←→ Last access time**
- **Which "old" cache line to replace? ("Replacement policy")**
  - Random
  - Most common: Least Recently Used (LRU)
  - Not recently used (NRU) combined with other criteria
- **"State" of cache line:**
  - Cache line has NOT been MODIFIED – valid copy of data in main memory → "Old" cache line can be overwritten with new data
  - MODIFIED: At least part of the cache line has been modified, so the copy of data in main memory is invalid → "Dirty" cache line needs to be written back ("evicted") to main memory before it can be overwritten in cache

Temporal Locality: Taking Account of Cache Size

- **Efficient use of caches requires "data reuse": data in the cache is reused several times before it is replaced…. ("Temporal locality")**

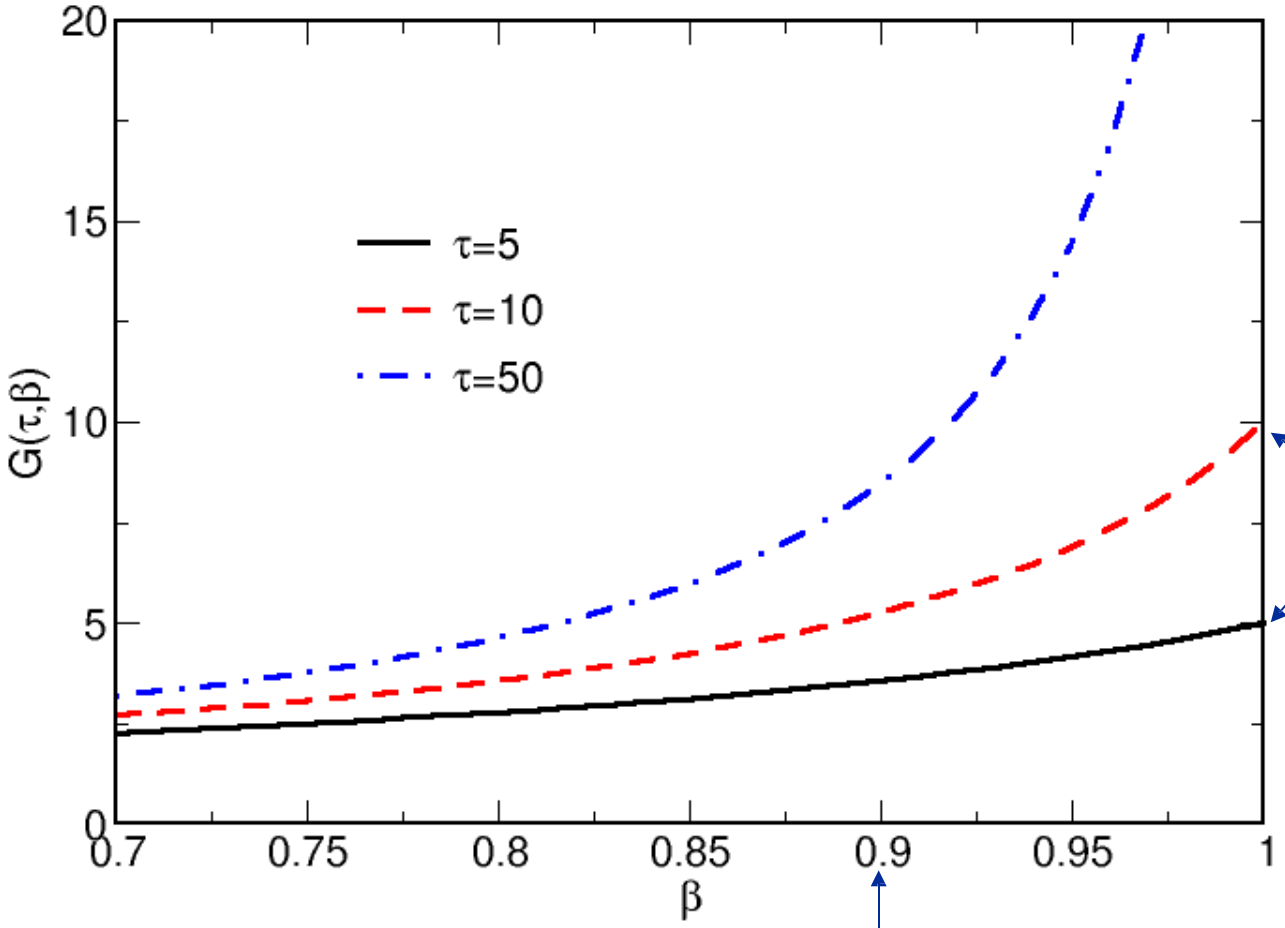
Assume large N

```
A(1:N)=B(1:N)+Z(1:N)                    DO I = 1,N
                                           A(I) = B(I)+Z(I)
C(1:N)=C(1:N)*Z(1:N)        ------>        C(I) = C(I)*Z(I)
                                           E(I) = Z(I)+A(I)*C(I)
E(1:N)=Z(1:N)+A(1:N)*C(1:N)             ENDDO
```

- **Assessing the potential benefit of using caches**
  - β : cache reuse ratio (fraction of loads/stores with cache hits)
  - Access time for main memory (cache): T_m (T_c); ratio: $\tau = T_m/T_c$
  - $T_{av}(\beta) = \beta * T_c + (1 - \beta) * T_m$ → $T_{av}(0) = T_m$ & $T_{av}(1) = T_c$

Performance Gain: $$G(\tau,\beta) = \frac{T_m}{T_{av}} = \frac{\tau T_c}{\beta T_c + (1-\beta)\tau T_c} = \frac{\tau}{\beta + \tau(1-\beta)}$$

Potential Gain from Temporal Locality



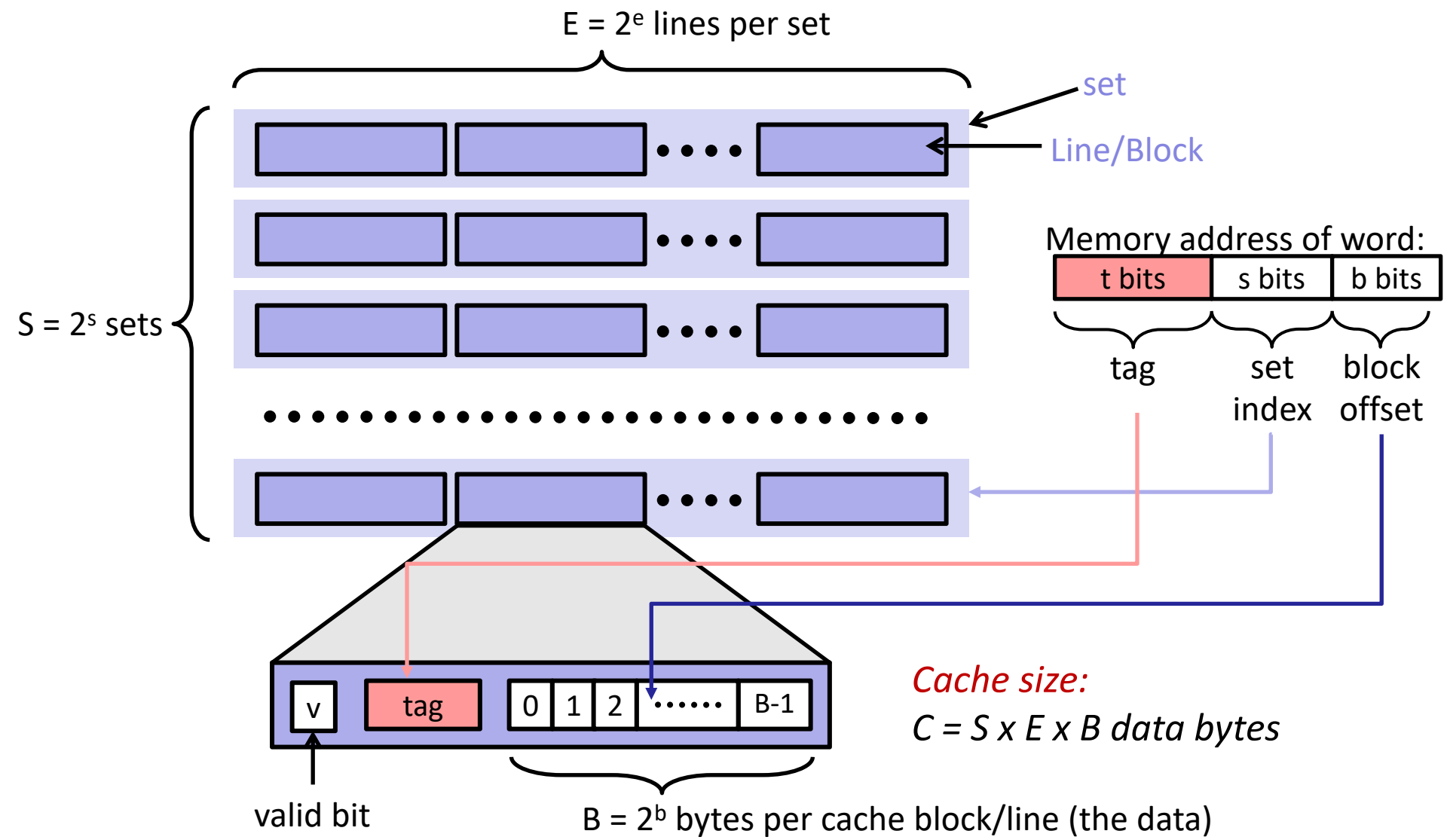
$G(\tau, 1) = \tau$

$\beta = 0.9 \rightarrow$ load once from main memory and reuse it 9 times!

Cache mappings

- Fully associative – a new line can be placed at any location in the cache (not common with large caches)
- Direct mapped – each cache line has a unique location in the cache to which it will be assigned (also not common)
- E-way set-associative – each cache line can be placed in one of E different locations in a specific “set” in the cache
  - In this case, the system needs to decide which line should be replaced or evicted when all E locations are in use

General Cache Organization (S, E, B)

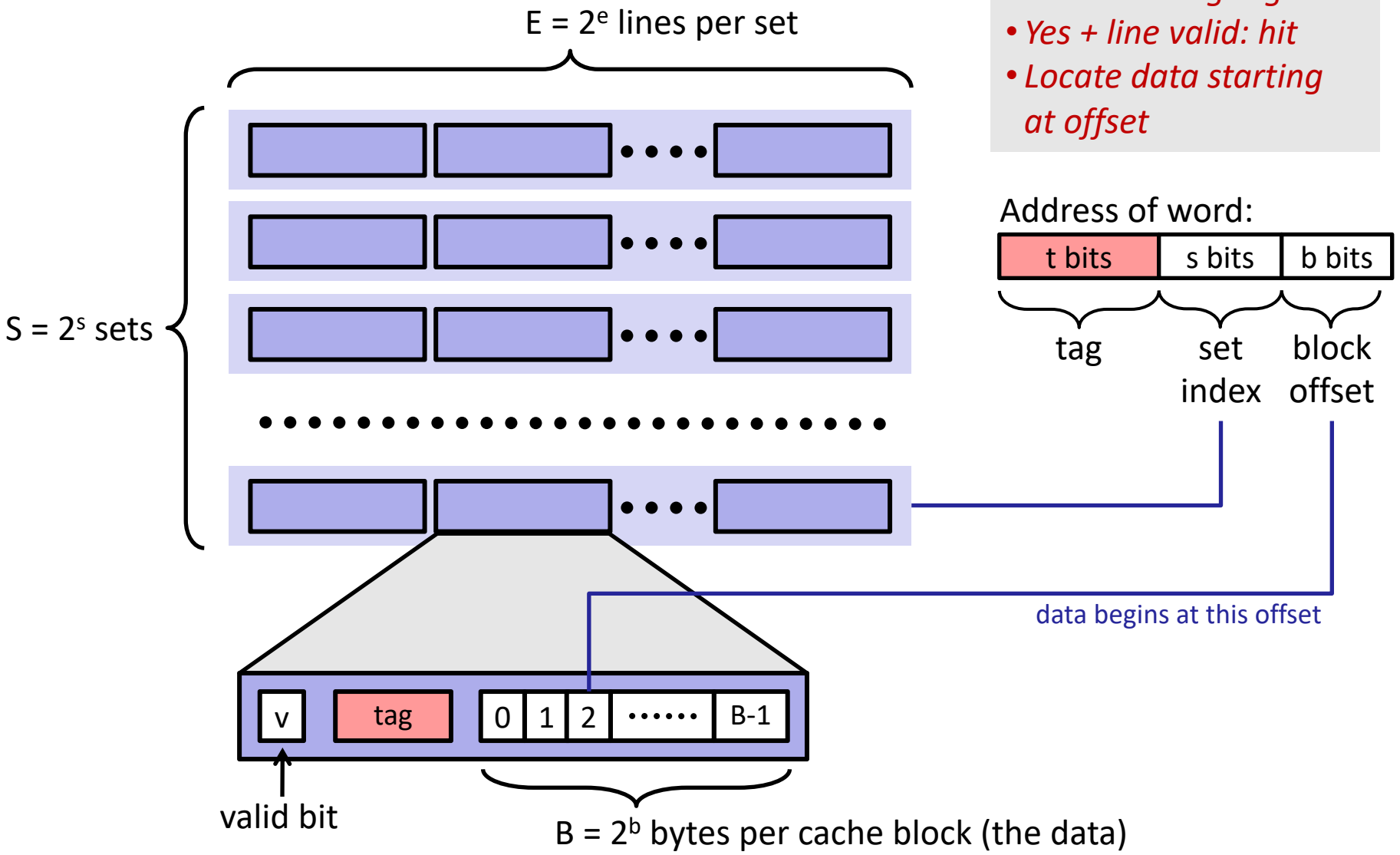


Cache size:

C = S x E x B data bytes

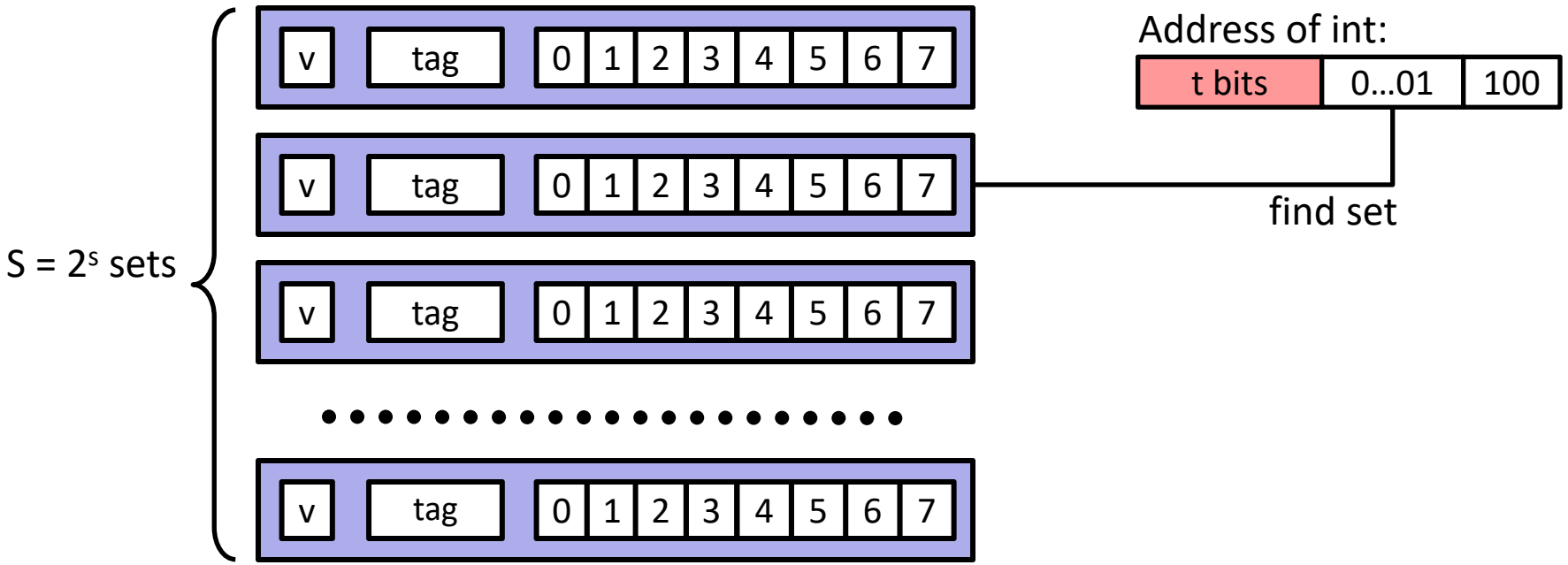
Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



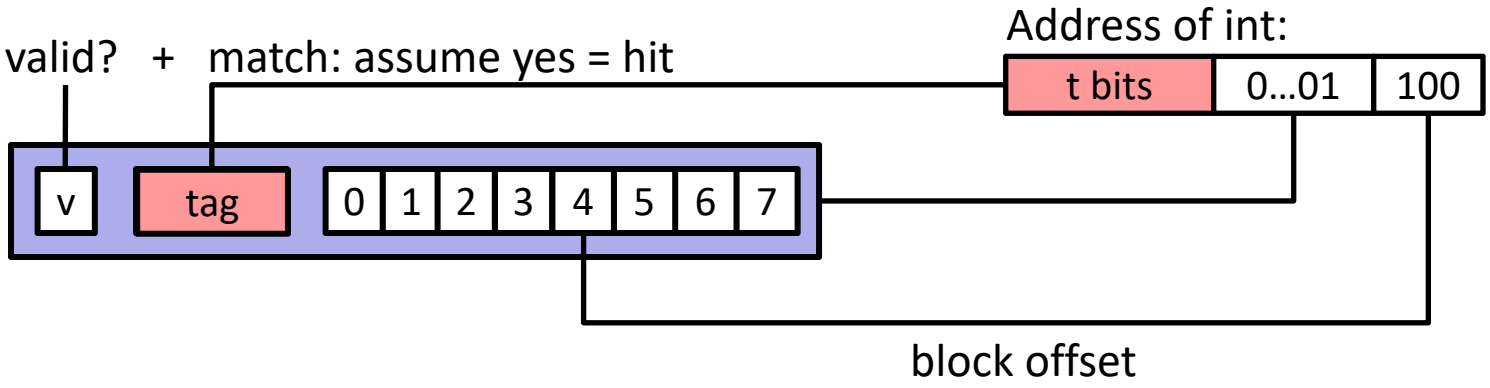
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



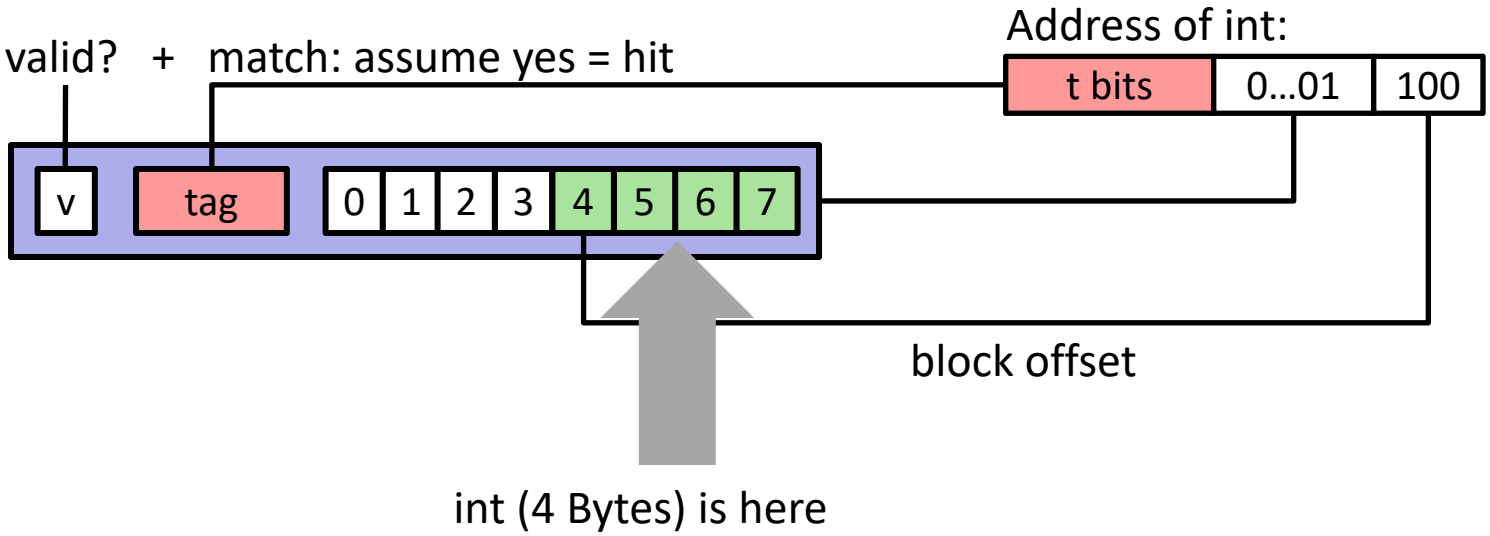
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes

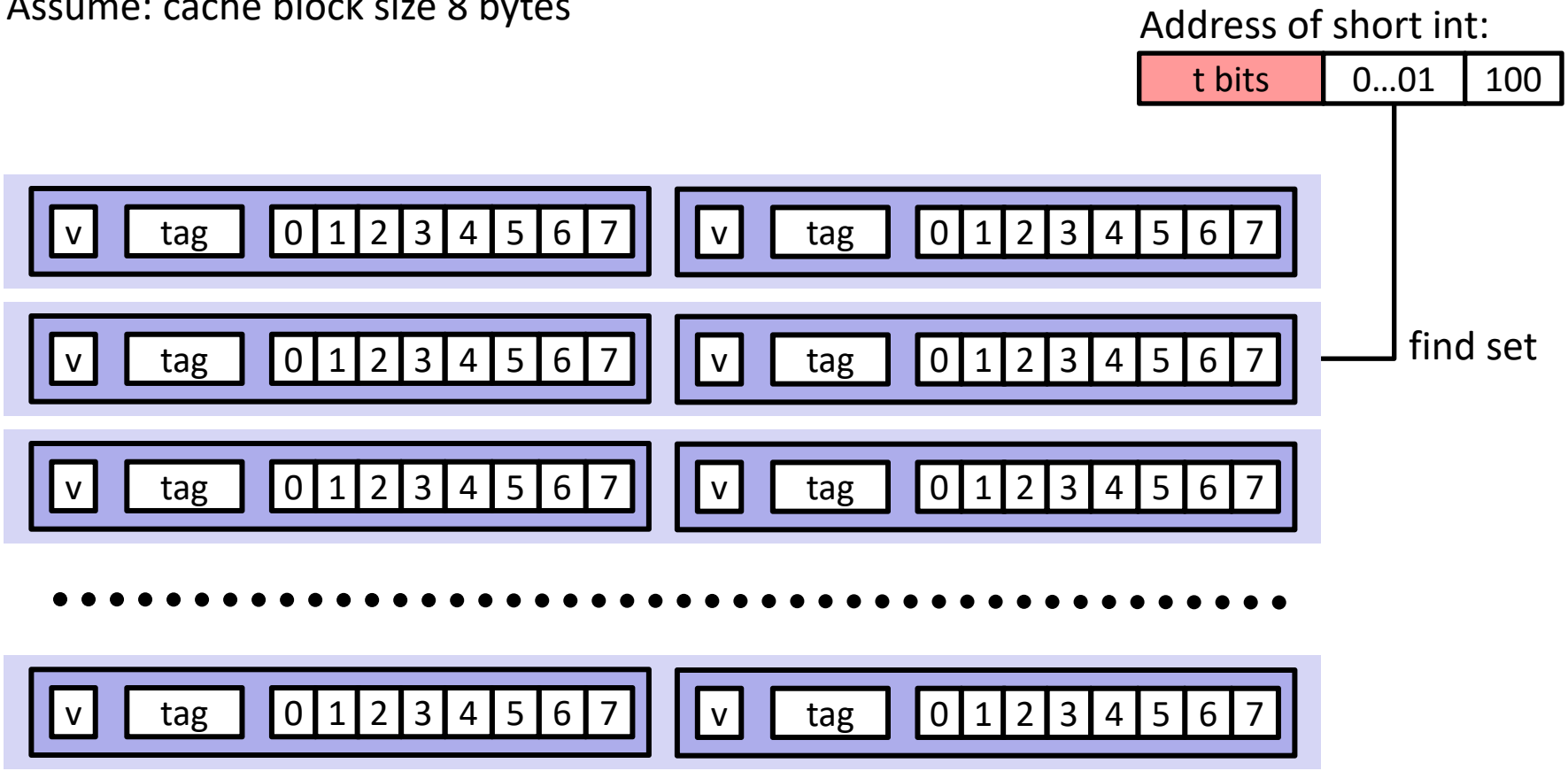


No match: old line is evicted and replaced

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

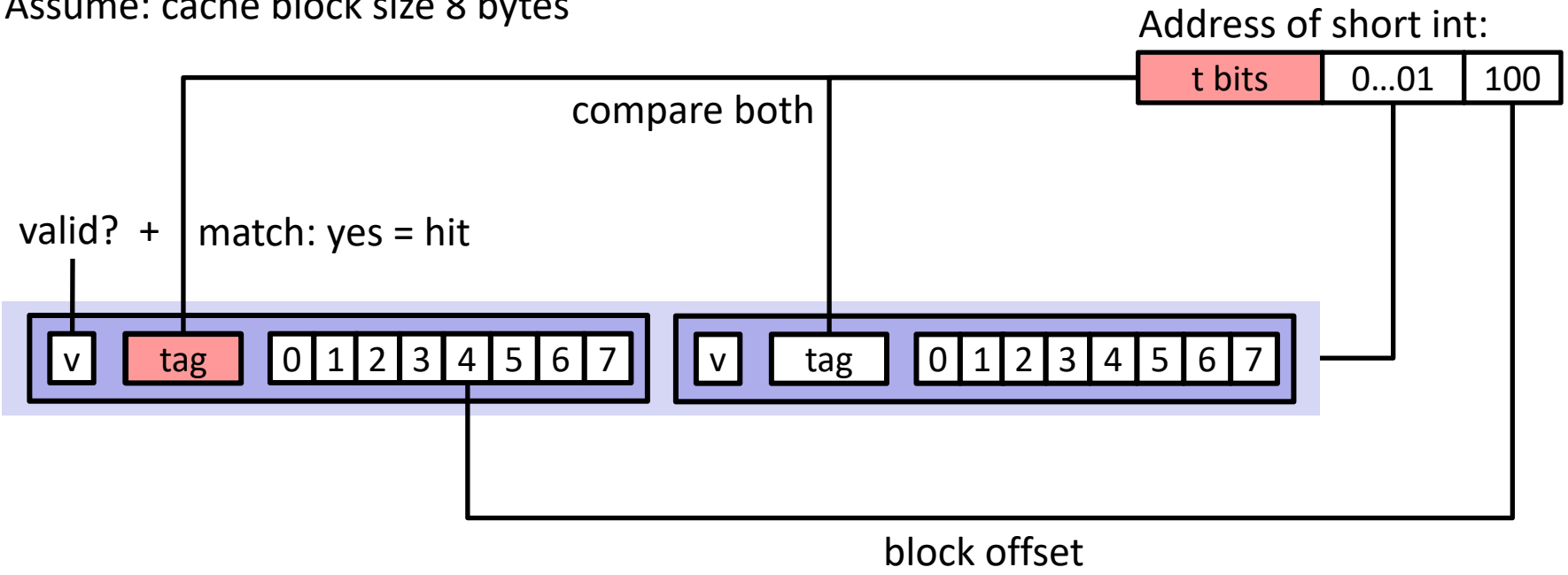
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

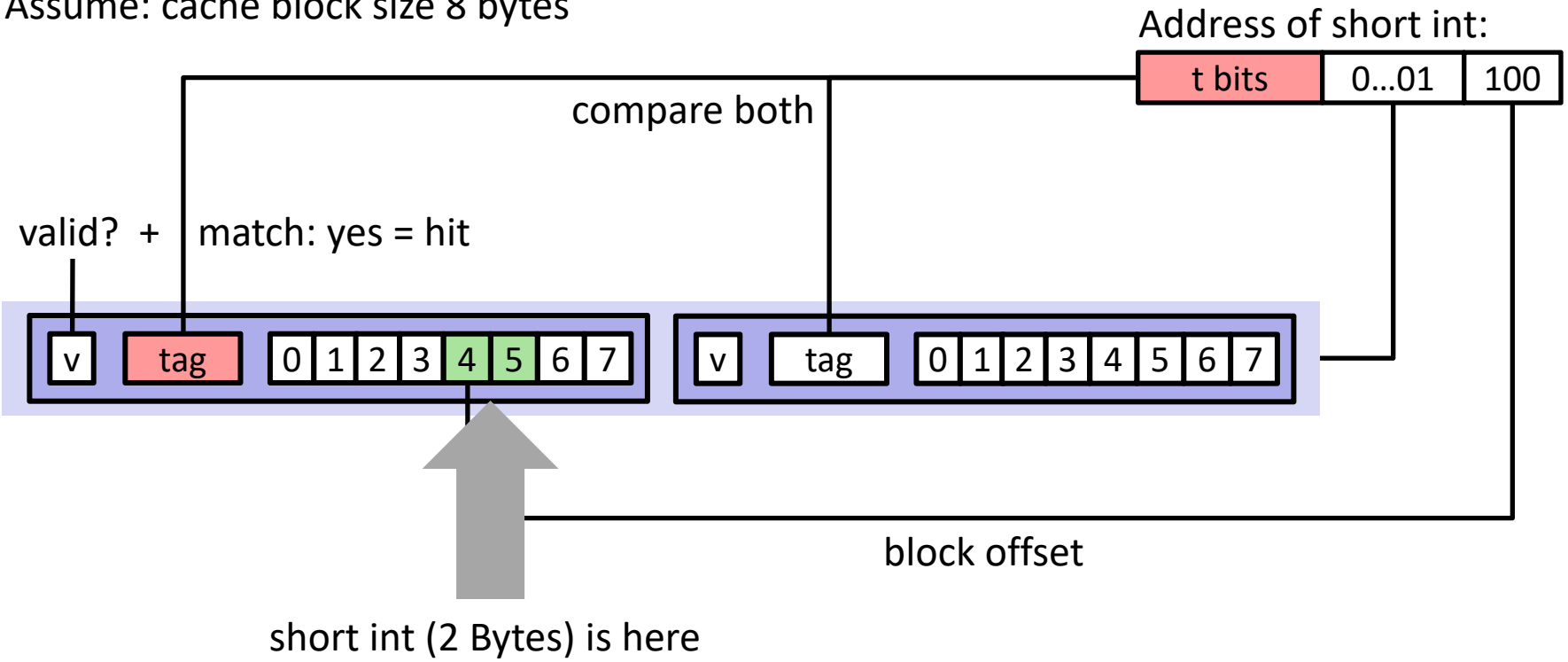
E = 2: Two lines per set

Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes



No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

What about writes?

- Multiple copies of data exist:
  - L1, L2, L3, Main Memory, Disk
  - Note: Nehalem L2 and L3 caches are “inclusive” caches, meaning that anything in Lk is also in Ln for n>k
- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need to add a “dirty bit” (line different from memory or not)
- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes immediately to memory)
- Typical
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate (Core i7/Nehalem)**

Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses / accesses) = 1 – hit rate
  - Typical numbers (in percentages):
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.
- Hit Time
  - Time to deliver a line in the cache to the processor
    - includes time to determine whether the line is in the cache
  - Typical numbers:
    - 1-2 clock cycle for L1
    - 5-20 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)

Lets think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory

- How much better is 99% hits than 97%?
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles

  - Average access time:
    97% hits: 1 cycle + 0.03 * 100 cycles = **4 cycles**
    99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles**

- This is why “miss rate” (3% vs. 1%) is used instead of “hit rate”