CS250P: Computer Systems Architecture Memory System and Caches

Sang-Woo Jun

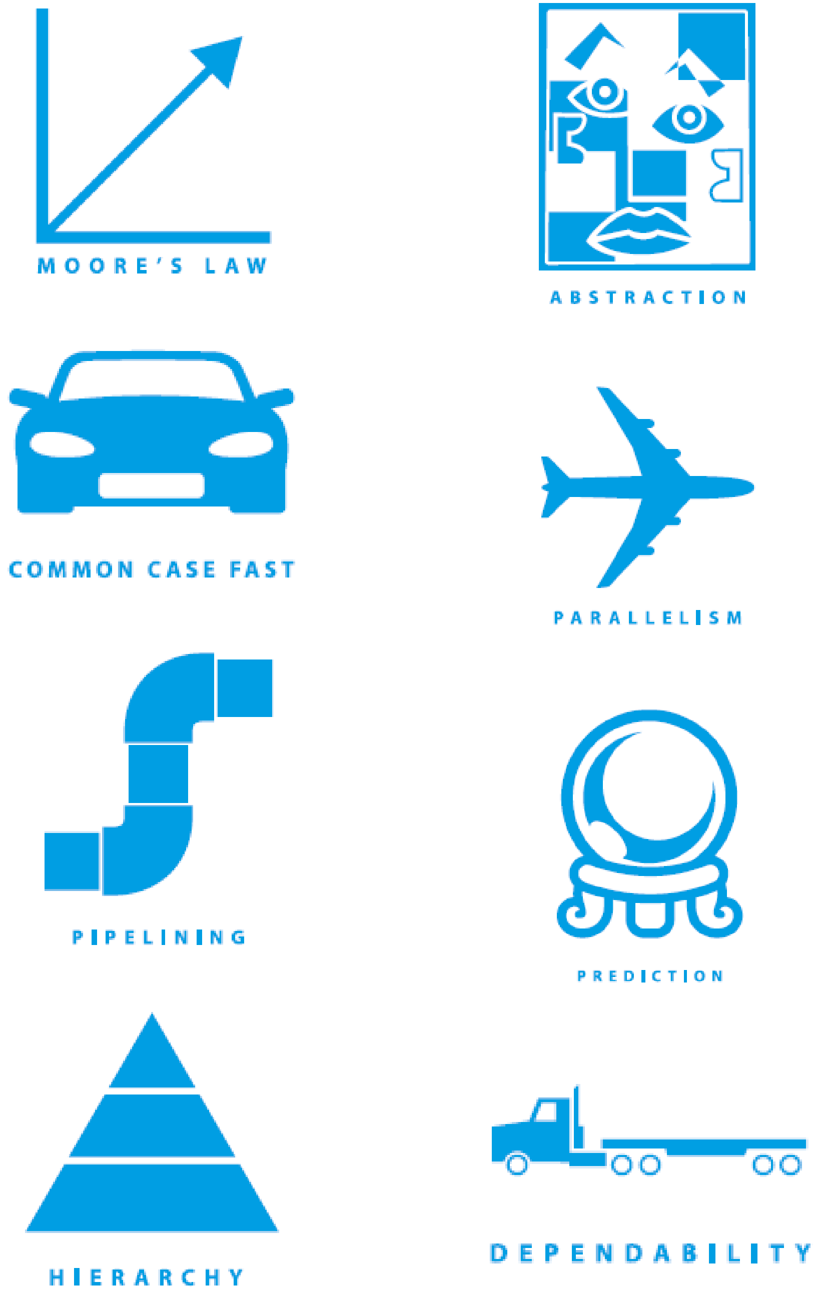
Fall 2022

UCI

Large amount of material adapted from MIT 6.004, “Computation Structures”,
Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”,
and CS 152 Slides by Isaac Scherson

Eight great ideas

- ❑ Design for Moore’s Law
- ❑ Use abstraction to simplify design
- ❑ Make the common case fast
- ❑ Performance via parallelism
- ❑ Performance via pipelining
- ❑ Performance via prediction
- ❑ Hierarchy of memories
- ❑ Dependability via redundancy



Caches are important

"There are only two hard things in computer science:

1. Cache invalidation,
2. Naming things,
3. and off-by-one errors"

Original quote (with only the first two points) by Phil Karlton
I couldn't find joke source

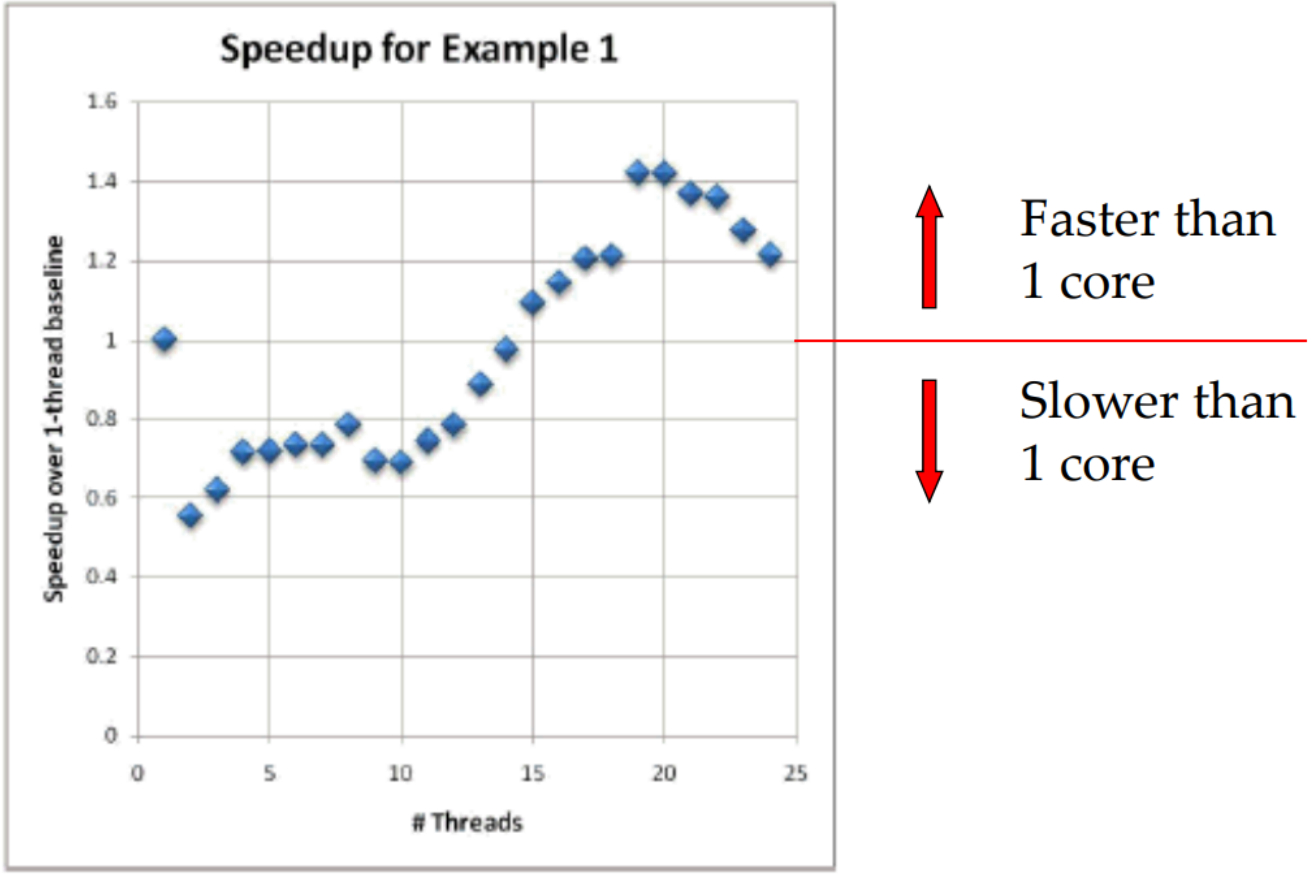
Motivation Example:
An Embarrassingly Parallel Workload

- A very simple example of counting odd numbers in a large array

```
int results[THREAD_COUNT];
void worker_thread(…) {
    int tid = …;
    for (e in myChunk) {
        if ( e % 2 != 0) results[tid]++;
    }
}
```

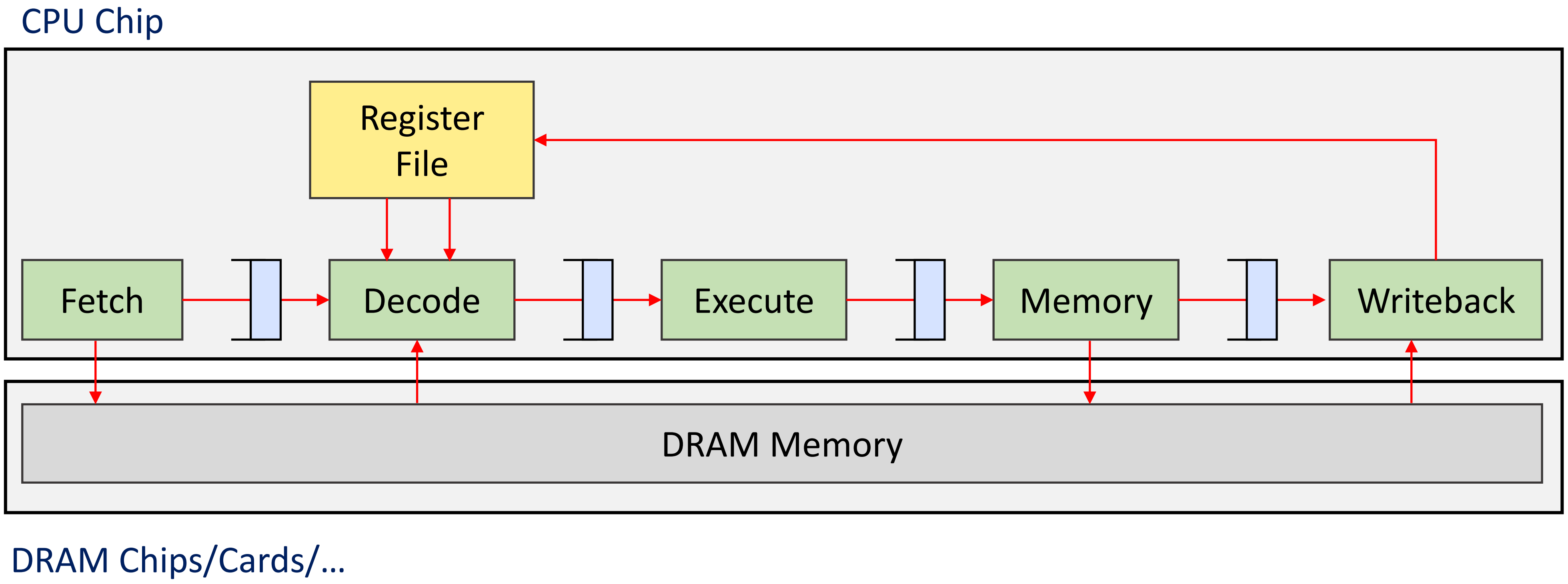
Do you see any performance red flags?

Scalability Unimpressive

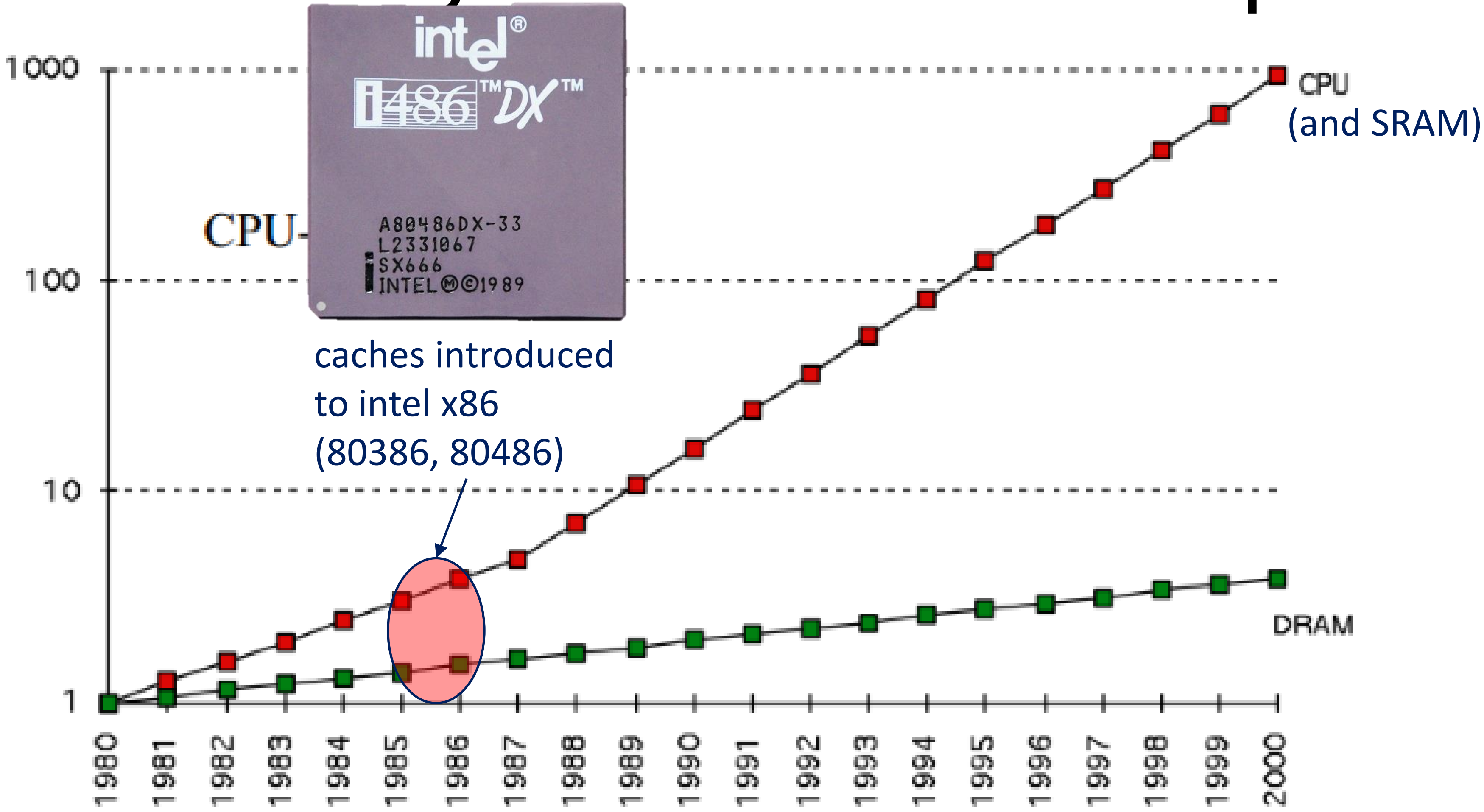


Scott Meyers, “CPU Caches and Why You Care,” 2013

Originally…


CPU Chip
Register
File
Fetch
Decode
Execute
Memory
Writeback
DRAM Memory
DRAM Chips/Cards/…

History of The Processor/Memory Performance Gap



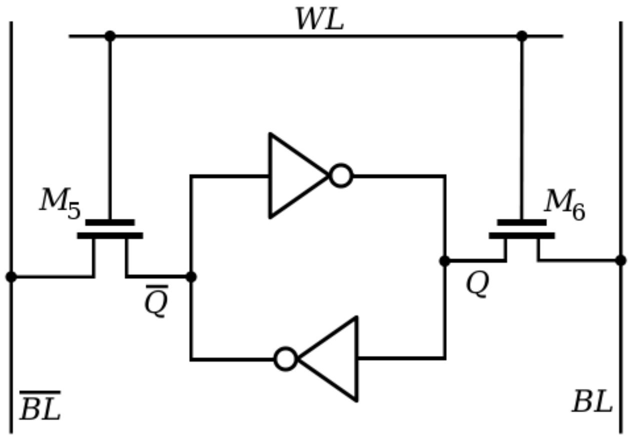
What is the Y-axis? Most likely normalized latency reciprocal

Source: Extreme tech, “How L1 and L2 CPU Caches Work, and Why They’re an Essential Part of Modern Chips,” 2018

What causes the cost/performance difference? – SRAM

- SRAM (Static RAM) vs. DRAM (Dynamic RAM)
- SRAM: Register File, Cache
  - Constructed entirely out of transistors , which processor logic is made of
  - As fast as the rest of the processor
  - Subject to propagation delay, etc, which makes large SRAM blocks expensive and/or slow

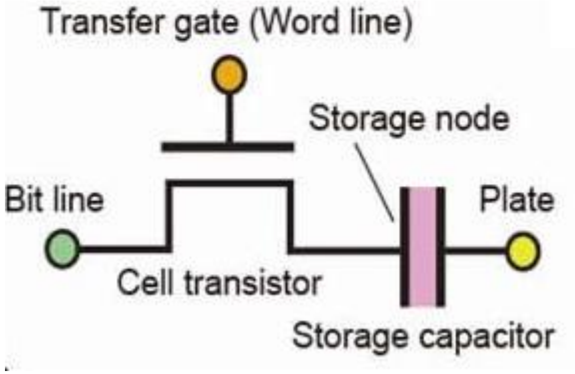
Size – performance trade-off necessary!

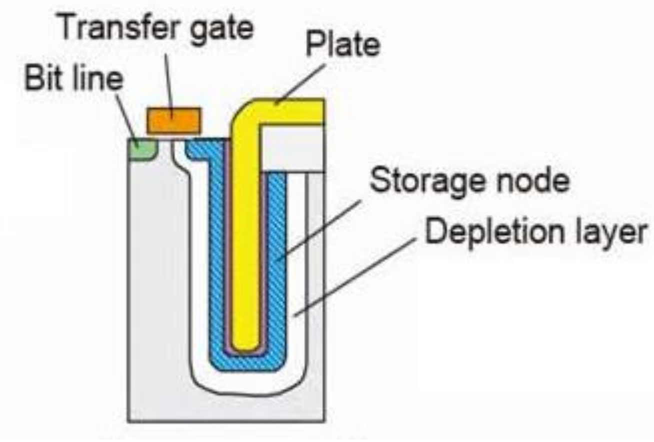


Source: Inductiveload, from commons.wikimedia.org

What causes the cost/performance difference? – DRAM

- DRAM stores data using a capacitor
  - Very small/dense cell
  - A capacitor holds charge for a short while, but slowly leaks electrons, losing data
  - To prevent data loss, a controller must periodically read all data and write it back ("Refresh")
    - Hence, "Dynamic" RAM
  - Requires fab process separate from processor
- Reading data from a capacitor is high-latency
  - EE topics involving sense amplifiers, which we won't get into



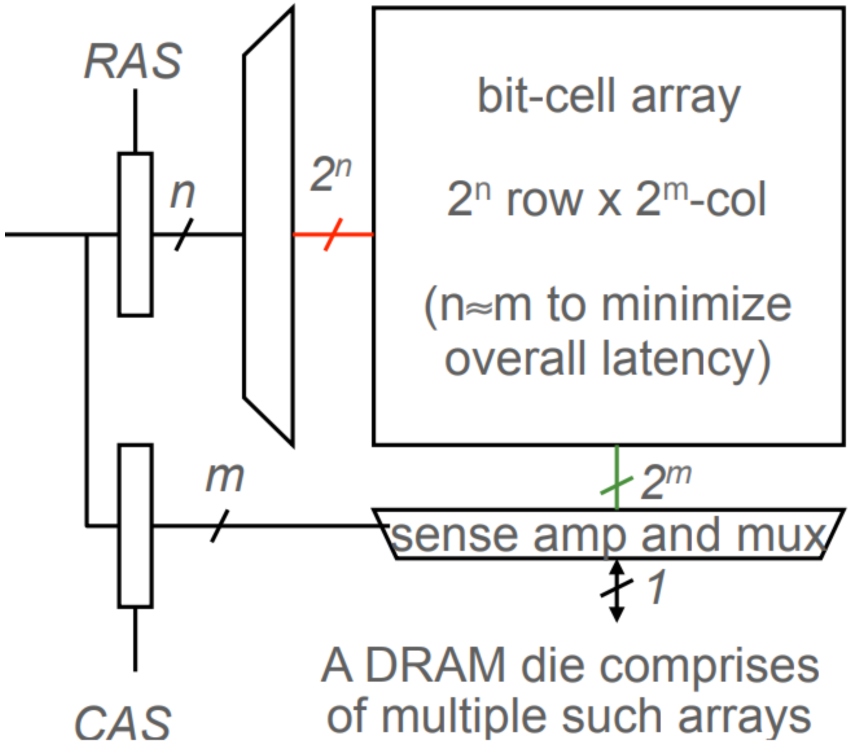


Note: Old, "trench capacitor" design

Source: Dailytech

What causes the cost/performance difference? – DRAM

- DRAM cells are typically organized into a rectangle (rows, columns)
  - Reduces addressing logic, which is a high overhead in such dense memory
  - Whole row must be read whenever data in new row is accessed
  - Right now, typical row size ~8 KB
- Fast when accessing data in same row, order of magnitude slower when accessing small data across rows
  - Accessed row temporarily stored in DRAM "row buffer"



Introducing caches

- The CPU is (largely) unaware of the underlying memory hierarchy
  - The memory abstraction is a single address space
  - The memory hierarchy <u>transparently</u> stores data in fast or slow memory, depending on usage patterns
- Multiple levels of “caches” act as interim memory between CPU and main memory (typically DRAM)
  - Processor accesses main memory (transparently) through the cache hierarchy
  - If requested address is already in the cache (address is “cached”, resulting in “cache hit”), data operations can be fast
  - If not, a “cache miss” occurs, and must be handled to return correct data to CPU

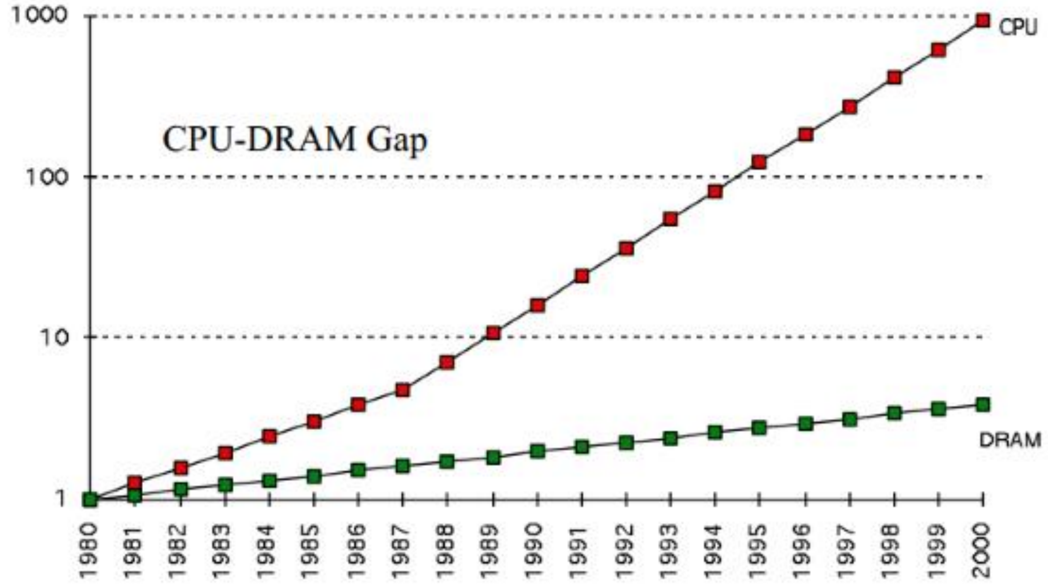
Caches Try to Be Transparent

- Software is (ideally) written to be oblivious to caches
  - Programmer should not have to worry about cache properties
  - Correctness isn't harmed regardless of cache properties

- However, the performance impact of cache affinity is quite high!
  - Performant software cannot be written in a completely cache-oblivious way

History of The Processor/Memory Performance Gap

- Processor vs Memory Performance



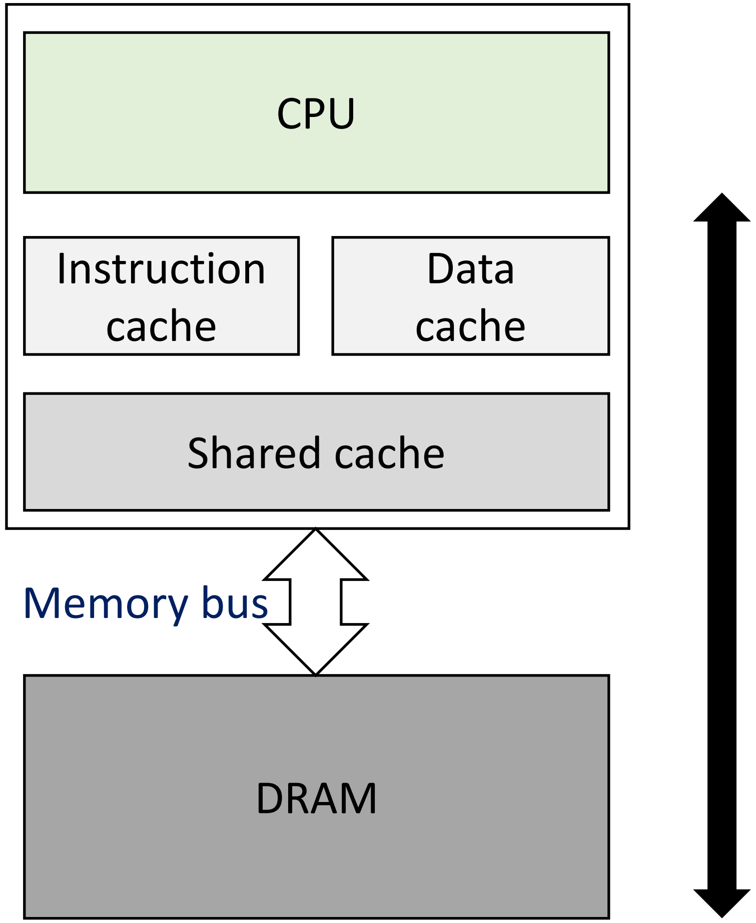
1980: no cache in microprocessor;
1995 2-level cache

What is the Y-axis? Most likely normalized latency reciprocal

- 80386 (1985) :
  Last Intel desktop CPU with no on-chip cache
  (Optional on-board cache chip though!)
- 80486 (1989) : 4 KB on-chip cache
- Coffee Lake (2017) :
  64 KiB L1 Per core
  256 KiB L2 Per core
  Up to 2 MiB L3 Per core (Shared)

Source: Extreme tech, “How L1 and L2 CPU Caches Work, and Why They’re an Essential Part of Modern Chips,” 2018

A modern computer has a hierarchy of memory



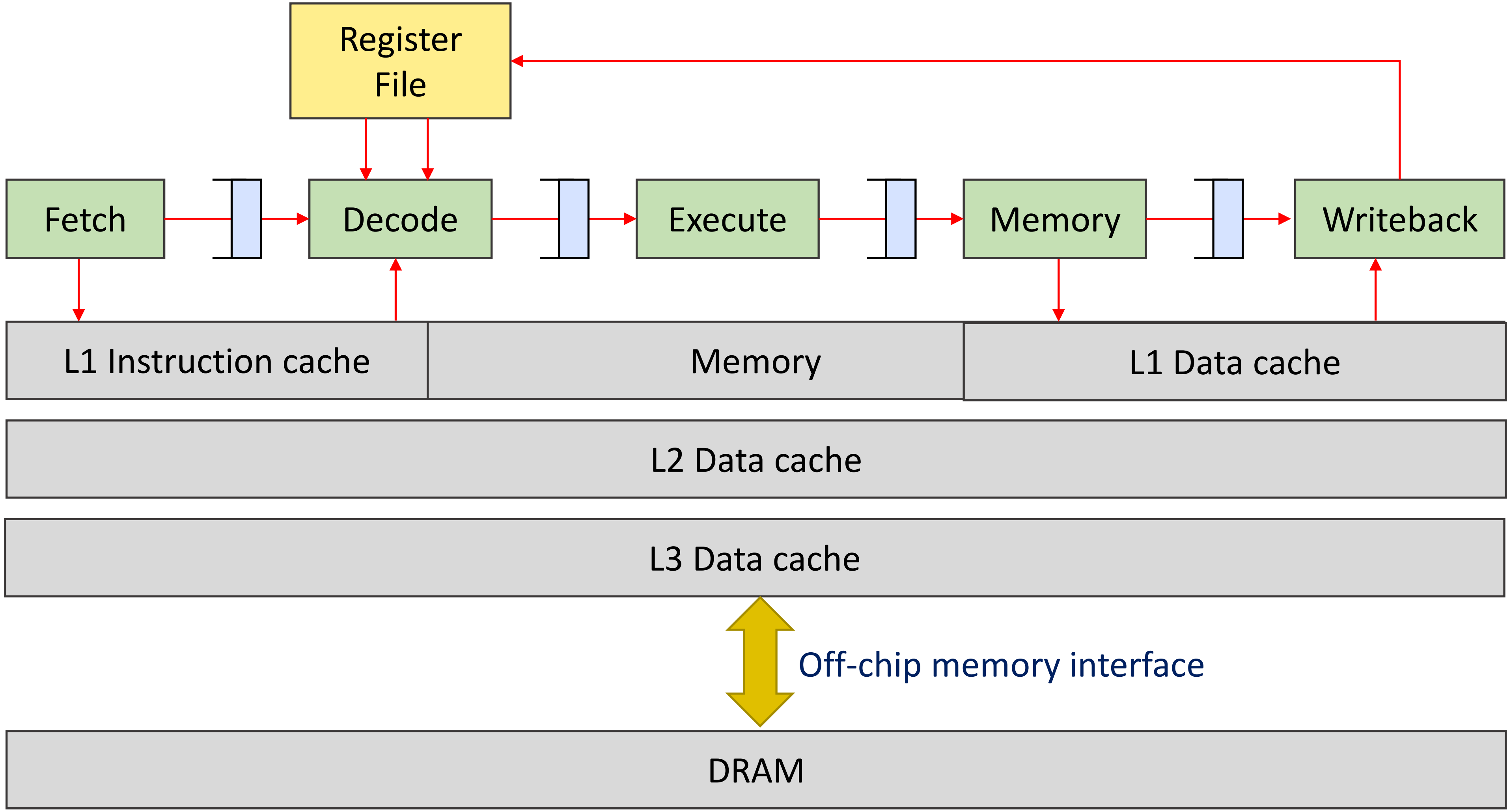
SRAM Caches
Low latency (~1 cycle)
Small (KBs)
Expensive ($1000s per GB)

DRAM
High latency (100s~1000s of cycles)
Large (GBs)
Cheap (<$5 per GB)

Cost prohibits having a lot of fast memory

Ideal memory:
As cheap and large as DRAM (Or disk!)
As fast as SRAM
…Working on it!

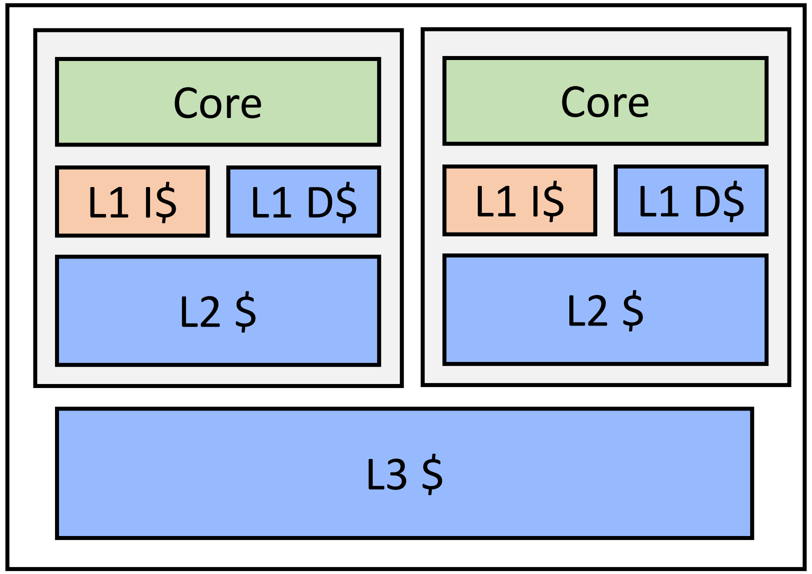
Caches and the processor pipeline



Multi-Layer Cache Architecture

Numbers from modern Xeon processors (Broadwell – Kaby lake)

Cache Level	Size	Latency (Cycles)
L1	64 KiB	< 5
L2	256 KiB	< 20
L3	~ 2 MiB per core	< 50

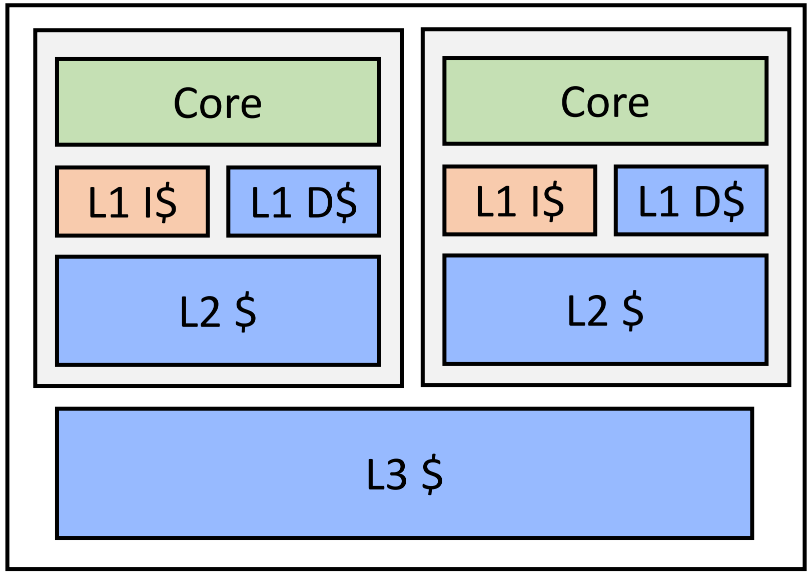


- Even with SRAM there is a size-performance trade-off
  - Not because the transistors are any different!
  - Cache management logic becomes more complicated with larger sizes
- L1 cache accesses can be hidden in the pipeline
  - Modern processors have pipeline depth of 14+
  - All others take a performance hit

Multi-Layer Cache Architecture

Numbers from modern Xeon processors (Broadwell – Kaby lake)

Cache Level	Size	Latency (Cycles)
L1	64 KiB	< 5
L2	256 KiB	< 20
L3	~ 2 MiB per core	< 50
DRAM	100s of GB	> 100*



- *This is in an ideal scenario
  - Actual measurements could be multiple hundreds or thousands of cycles!
- DRAM systems are complicated entities themselves
  - Latency/Bandwidth of the same module varies immensely by situation…

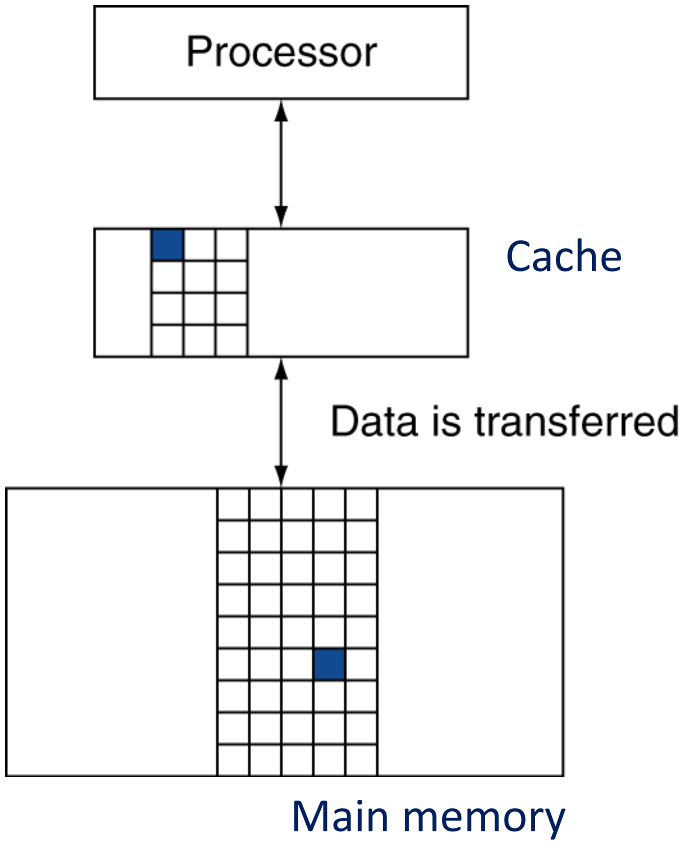
Cache operation

- One of the most intensely researched fields in computer architecture
- Goal is to *somehow* make to-be-accessed data available in fastest possible cache level at access time
  - Method 1: Caching recently used addresses
    - Works because software typically has **"Temporal Locality"** : If a location has been accessed recently, it is likely to be accessed (reused) soon
  - Method 2: Pre-fetching based on future pattern prediction
    - Works because software typically has **"Spatial Locality"** : If a location has been accessed recently, it is likely that nearby locations will be accessed soon
  - Many, many more clever tricks and methods are deployed!

Basic cache operations

- Unit of caching: “Block” or “Cache line”
  - **May be multiple words** -- 64 Bytes in modern Intel x86
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
  - Then accessed data supplied from upper level

How does the memory system keep track of what is present in cache?

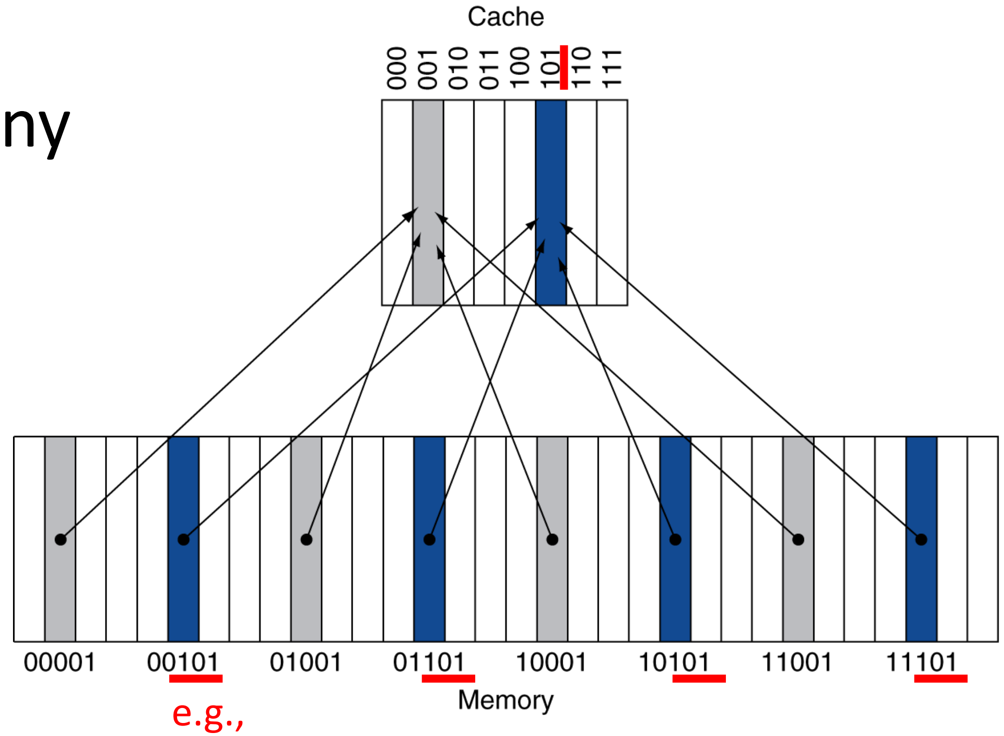


A simple solution: “Direct Mapped Cache”

- ❑ Cache location determined by address
- ❑ Each block in main memory mapped on one location in cache memory (“Direct Mapped”)
  - ○ “Direct mapped”
- ❑ Cache is smaller than main memory, so many DRAM locations map to one cache location

(Cache $\text{address}_{\text{block}}$)
= (main memory $\text{address}_{\text{block}}$) mod (cache $\text{size}_{\text{block}}$)

Since cache size is typically power of two,
Cache address is lower bits of block address



Selecting index bits

- Why do we chose low order bits for index?
    - Allows consecutive memory locations to live in the cache simultaneously
        - e.g., 0x00**01** and 0x00**02** mapped to different slots
    - Reduces likelihood of replacing data that may be accessed again in the near future
    - Helps take advantage of locality

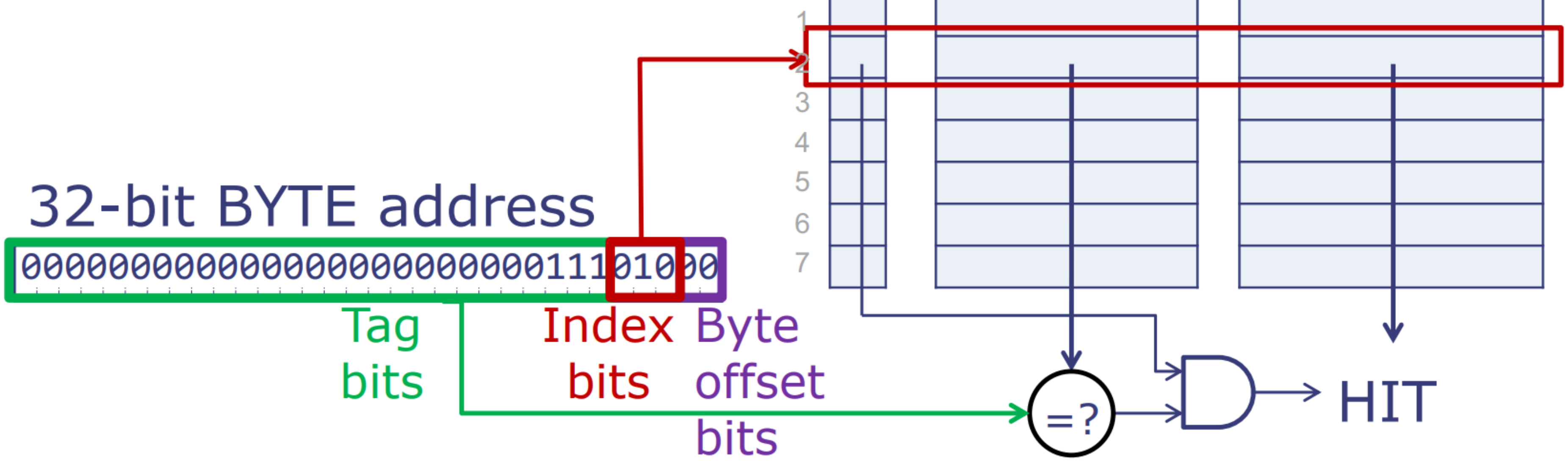
Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data, compare when read
  - Actually, only need the high-order bits (Called the “tag”)
- What if there is a cache slot is still unused?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

Direct Mapped Cache Access

- For cache with 2^W cache lines
  - Index into cache with W address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, cache hit!

Example 8-line direct-mapped cache:



Direct-Mapped Cache Problem: Conflict Misses

- Assuming a 1024-line direct-mapped cache, 1-word cache line
- Consider steady state, after already executing the code once
  - What can be cached has been cached
- Conflict misses:
  - Multiple accesses map to same index!

Loop A: Code at 1024, data at 37

Word Address	Cache Line index	Hit/ Miss
1024	0	HIT
37	37	HIT
1025	1	HIT
38	38	HIT
1026	2	HIT
39	39	HIT
1024	0	HIT
37	37	HIT
...		

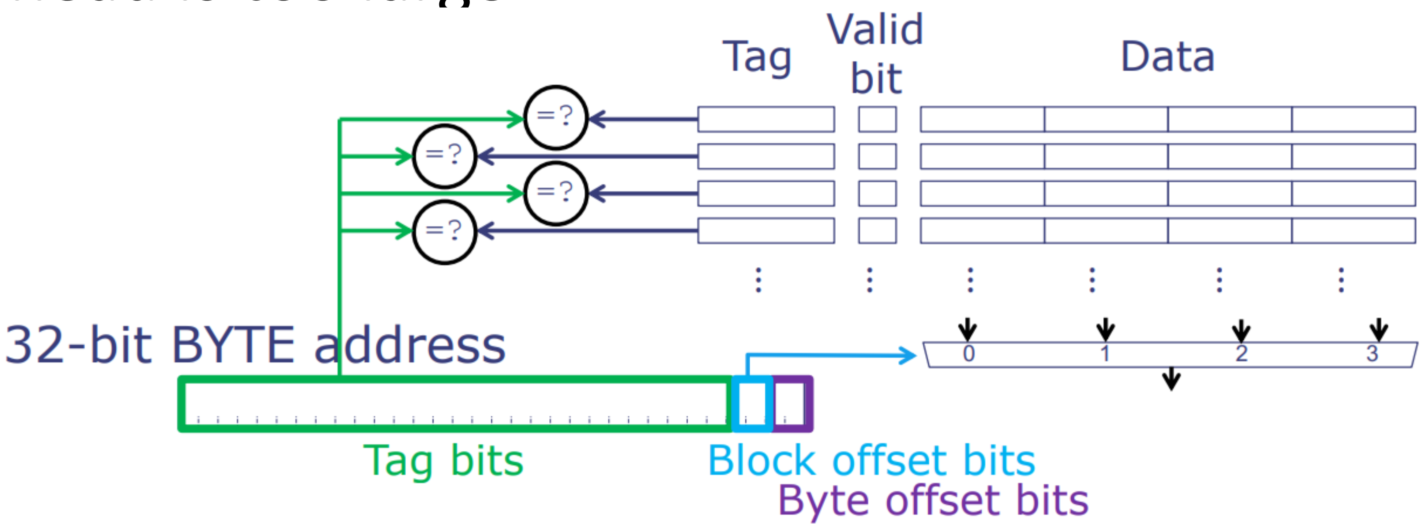
Loop B: Code at 1024, data at 2048

Word Address	Cache Line index	Hit/ Miss
1024	0	MISS
2048	0	MISS
1025	1	MISS
2049	1	MISS
1026	2	MISS
2050	2	MISS
1024	0	MISS
2048	0	MISS
...		

We have enough cache capacity, just inconvenient access patterns

Other extreme: "Fully associative" cache

- Any address can be in any location
  - No cache index!
  - Flexible (no conflict misses)
  - Expensive: Must compare tags of all entries in parallel to find matching one
- Best use of cache space (all slots will be useful)
- But management circuit overhead is too large



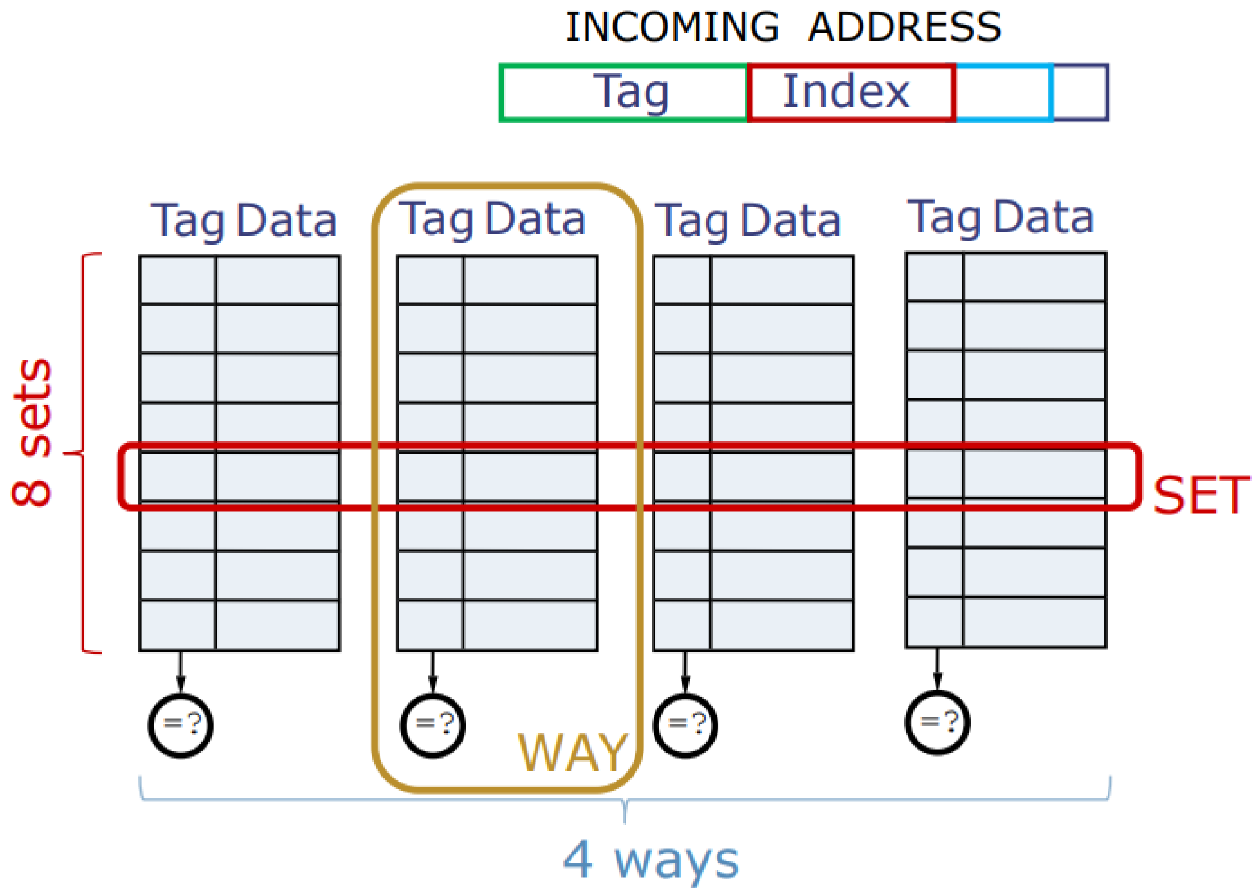
Three types of misses

- Compulsory misses (aka cold start misses)
  - First access to a block
- Capacity misses
  - Due to finite cache size
  - A replaced block is later accessed again
- Conflict misses (aka collision misses)
  - Conflicts that happen even when we have space left
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size

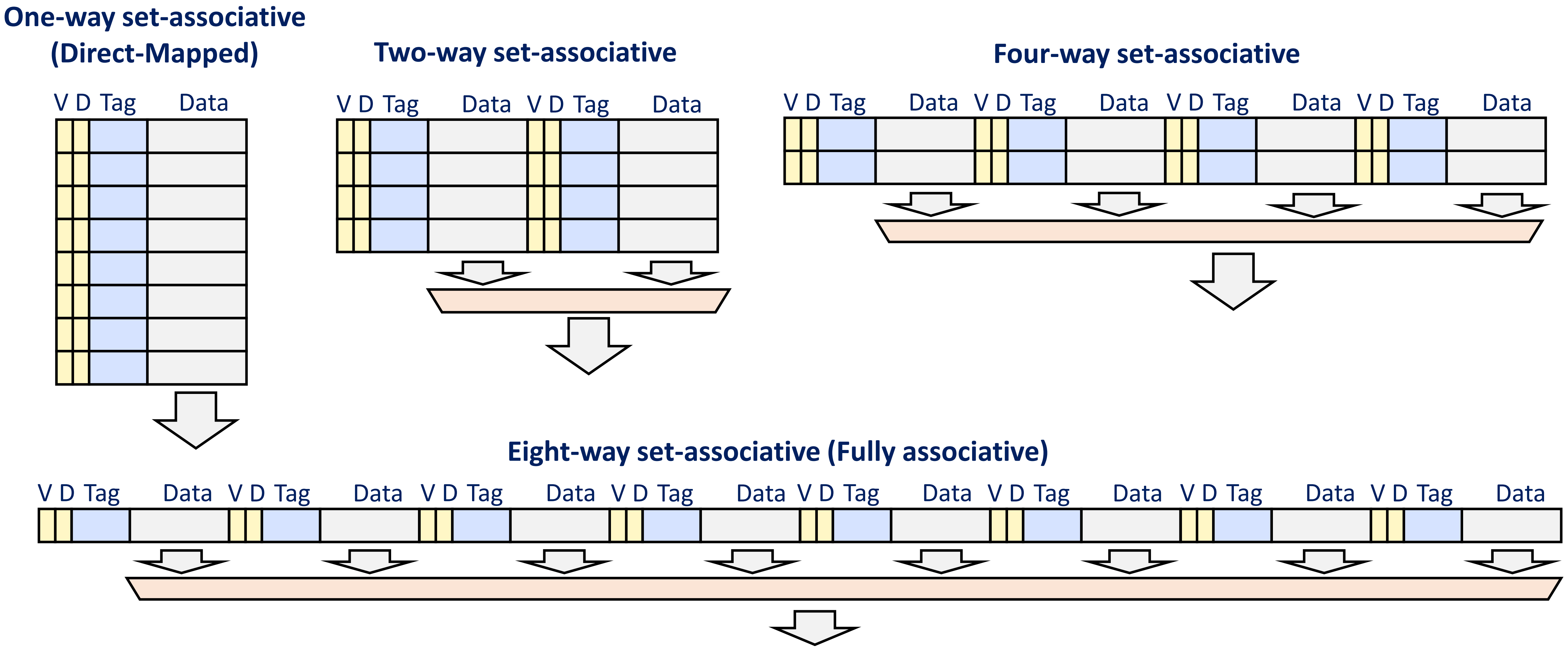
    Empty space can always be used in a fully associative cache
    (e.g., 8 KiB data, 32 KiB cache, but still misses? Those are conflict misses)

Balanced solution:
N-way set-associative cache

- Use multiple direct-mapped caches in parallel to reduce conflict misses
- Nomenclature:
  - # Rows = # Sets
  - # Columns = # Ways
  - Set size = #ways = “set associativity” (e.g., 4-way -> 4 lines/set)
- Each address maps to only one set, but can be in any way within the set
- Tags from all ways are checked in parallel



Spectrum of associativity (For eight total blocks)



Each “Data” is a cache line (~64 bytes), needs another mux layer to get actual word

Associativity example

- Compare caches with four elements
  - Block access sequence: 0, 8, 0, 6, 8

- Direct mapped (Cache index = address mod 4)

Time ↓

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Associativity example

- 2-way set associative (Cache index = address mod 2)

Time ↓

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

- Fully associative (No more cache index!)

Time ↓

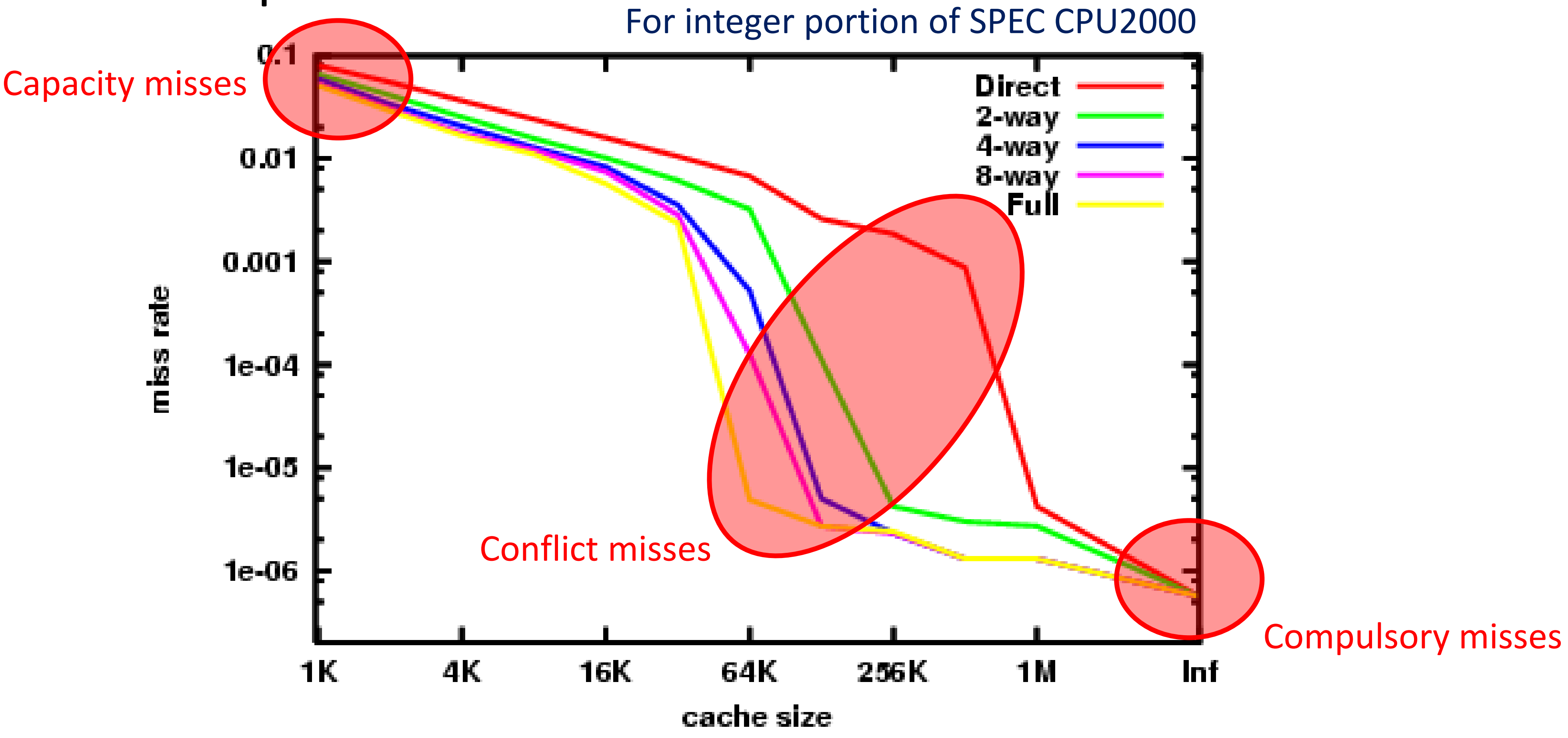
Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

How Much Associativity?

- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

How much associativity, how much size?

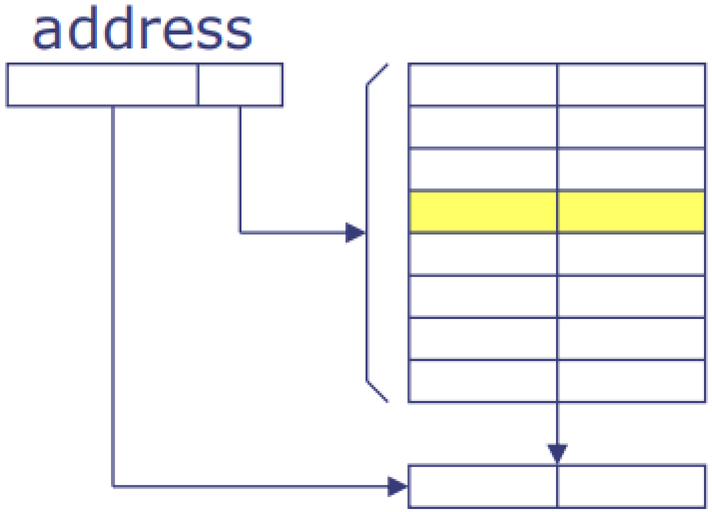
- Highly application-dependent!



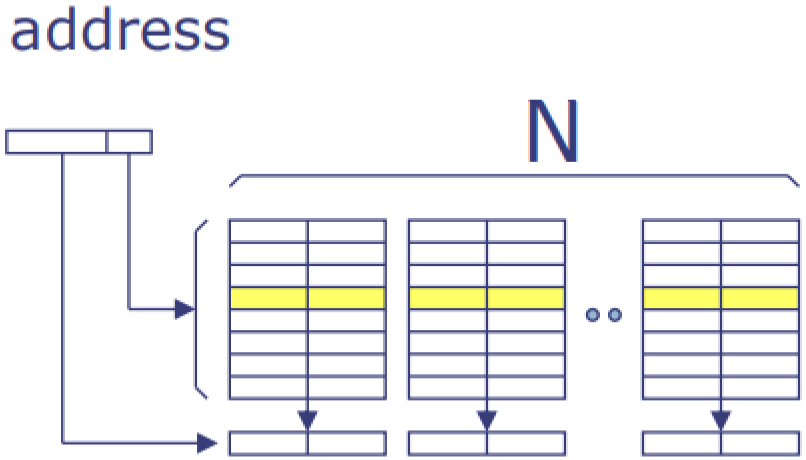
Piscione Pietro and Villardita Alessio, “Coherence and consistency models in multiprocessor architecture,” University of Pisa Computer Architecture, 2015

Associativity implies choice during misses

Direct-mapped



N-way set-associative



Only one place an address can go
In case of conflict miss, old data is simply evicted

Multiple places an address can go
In case of conflict miss, which way should we evict?

What is our "***replacement policy***"?

Replacement policies

- Optimal policy (Oracle policy):
  - Evict the line accessed furthest in the future
  - Impossible: Requires knowledge of the future!
- Idea: Predict the future from looking at the past
  - If a line has not been used recently, it's often less likely to be accessed in the near future (temporal locality argument)
- ***Least Recently Used (LRU):*** Replace the line that was accessed furthest in the past
  - Works well in practice
  - Needs to keep track of ordering, and discover oldest line quickly

Pure LRU requires complex logic: Typically implements cheap approximations of LRU

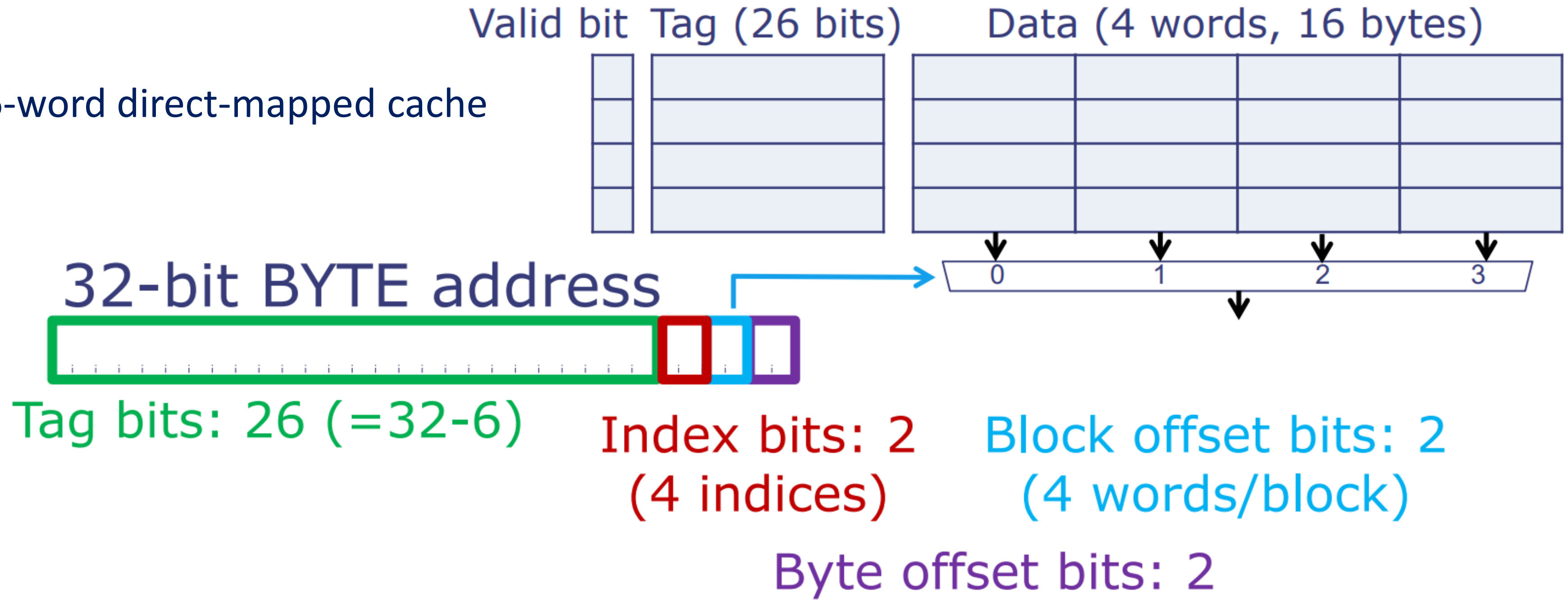
Other replacement policies

- LRU becomes very bad if working set becomes larger than cache size
  - "for (i = 0 to 1025) A[i];", if cache is 1024 elements large, every access is miss
- Some alternatives exist
  - Effective in limited situations, but typically not as good as LRU on average
  - Most recently used (MRU), First-In-First-Out (FIFO), random, etc …
  - Sometimes used together with LRU

Larger block (cache line) sizes

- Take advantage of spatial locality: Store multiple words per data line
  - Always fetch entire block (multiple words) from memory
  - Another advantage: Reduces size of tag memory!
  - Disadvantage: Fewer indices in the cache -> Higher miss rate!

Example: 4-block, 16-word direct-mapped cache



Cache miss with larger block

- 64 elements with block size == 4 words
  - 16 cache lines, 4 index bits
- Write 0x9 to 0x483C
  - 0100 1000 0011 1100

    Tag: 0x48 Index: 0x3 **-> Cache hit!**

    Block offset: 0x3
- Write 0x1 to 0x4938
  - 0100 1001 0011 1000

    Tag: 0x49 Index: 0x3 **-> Cache miss!**

    Block offset: 0x2

	V	D	Tag	Data			
0	1	1					
1	1	0					
2	0	0					
3	1	1	0x48				0x9
⋮							
15	0	0					

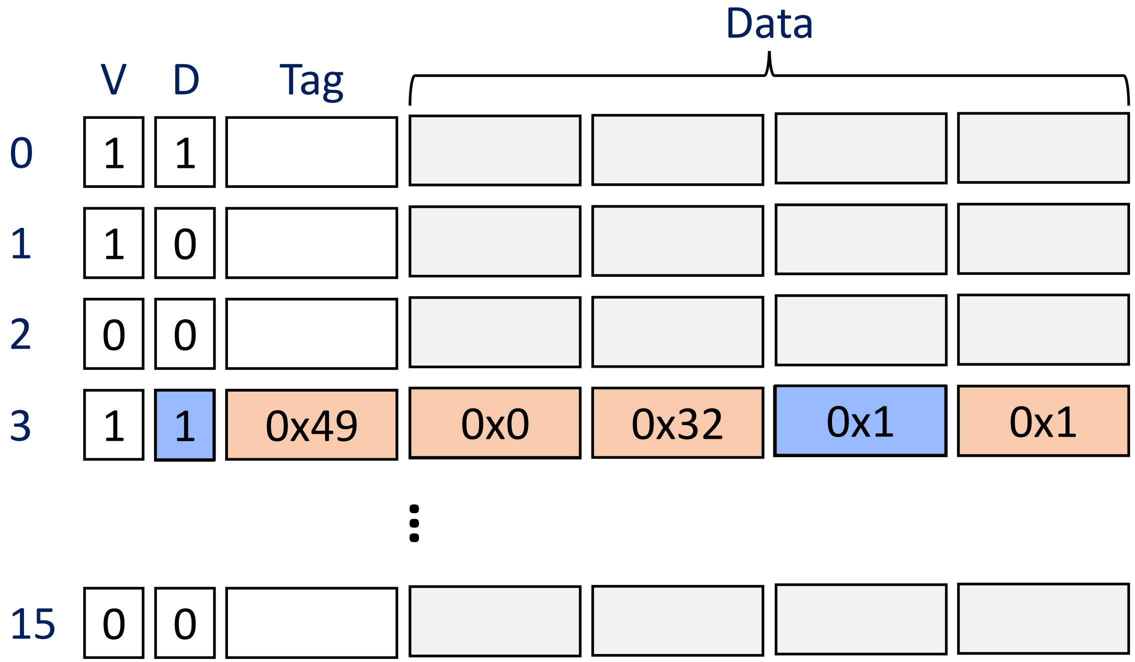
Cache miss with larger block

- Write 0x1 to 0x4938
  - 0100 1001 0011 1000

Tag: 0x49 Index: 0x3

Block offset: 0x2

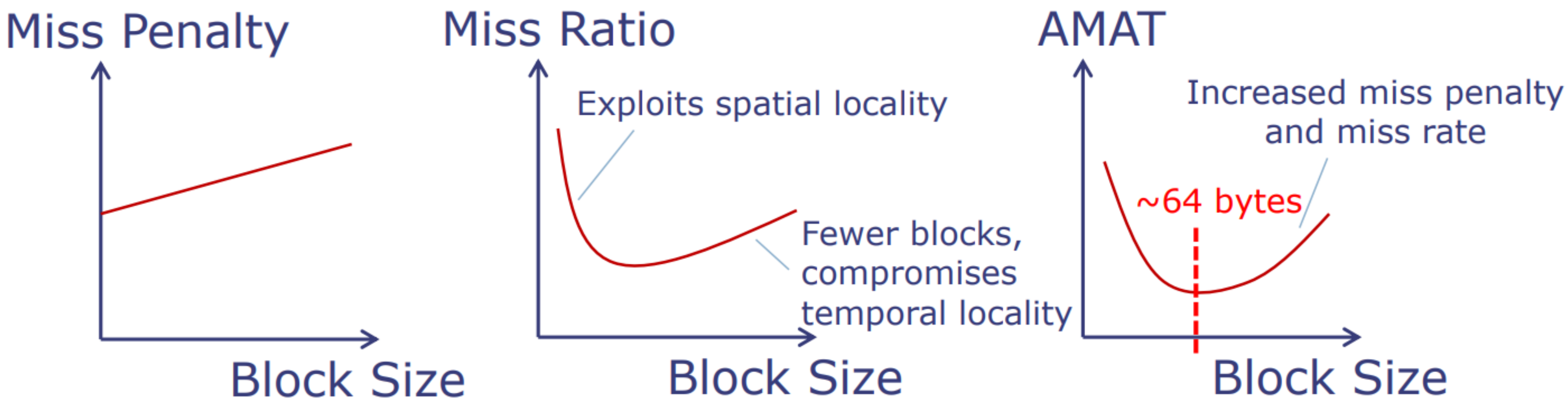
- Since D == 1,
  - Write cache line 3 to memory (All four words)
  - Load cache line from memory (All four words)
  - Apply write to cache



Writes/Reads four data elements just to write one!

Block size trade-offs

- Larger block sizes…
  - Take advantage of spatial locality (also, DRAM is faster with larger blocks)
  - Incur larger miss penalty since it takes longer to transfer the block from memory
  - Can increase the average hit time and miss ratio
- AMAT (Average Memory Access Time) = HitTime+MissPenalty*MissRatio



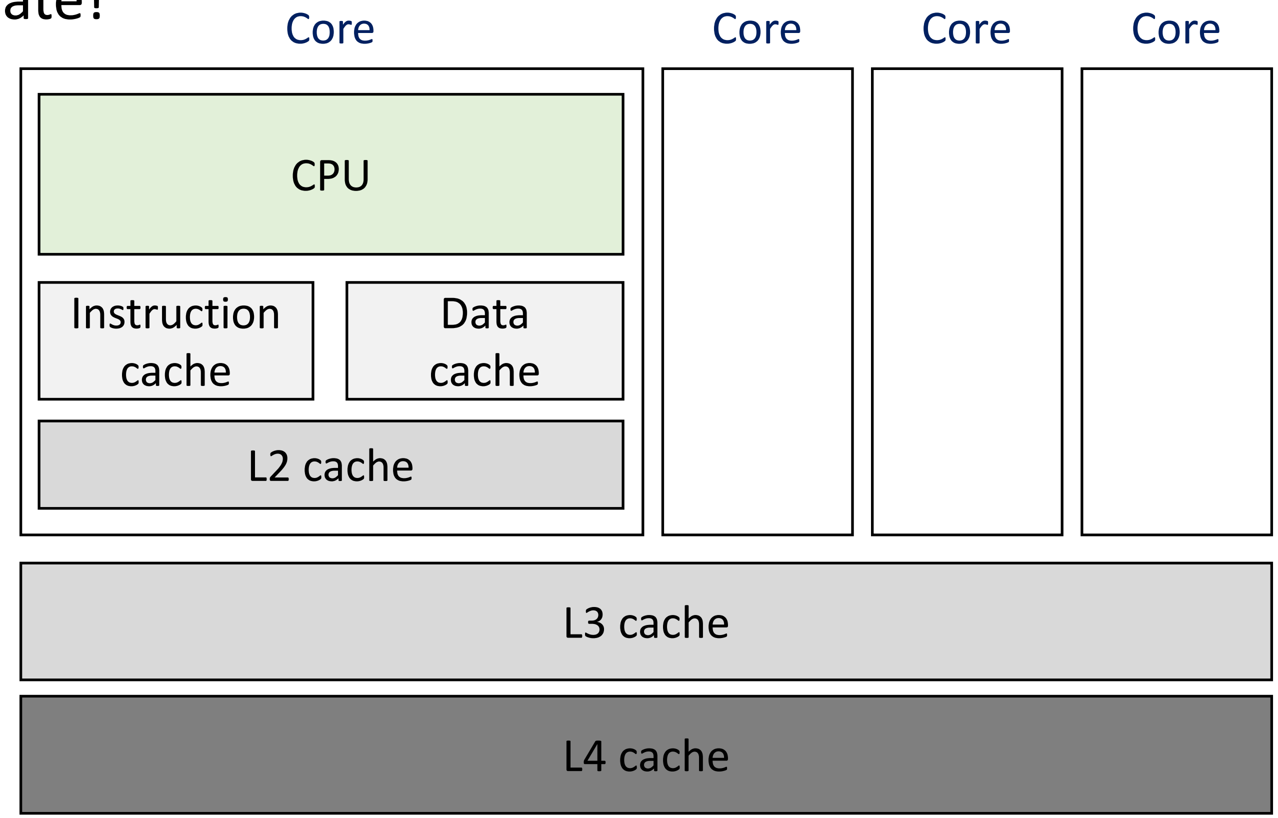
Performance improvements with caches

- Given CPU of CPI = 1, clock rate = 4GHz
  - Main memory access time = 100ns
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - CPI without cache = 400
- Given first-level cache with no latency, miss rate of 2%
  - Effective CPI = $1 + 0.02 \times 400 = 9$
- Adding another cache (L2) with 5ns access time, miss rate of 0.5%
  - Miss penalty = 5ns/0.25ns = 20 cycles
  - New CPI = $1 + 0.02 \times 20 + 0.005 \times 400 = 3.4$

	Base	L1	L2
CPI Improvements	400	9	3.4
IPC improvements	0.0025	0.11	0.29
Normalized performance	1	44	118

Real-world: Intel Haswell i7

- Four layers of caches (two per-core layers, two shared layers)
  - Larger caches have higher latency
  - Want to achieve both speed and hit rate!
- The layers
  - L1 Instruction & L1 Data:
    32 KiB, 8-way set associative
  - L2: 256 KiB, 8-way set associative
  - L3: 6 MiB, 12-way set associative
  - L4: 128 MiB, 16-way set associative
    eDRAM!

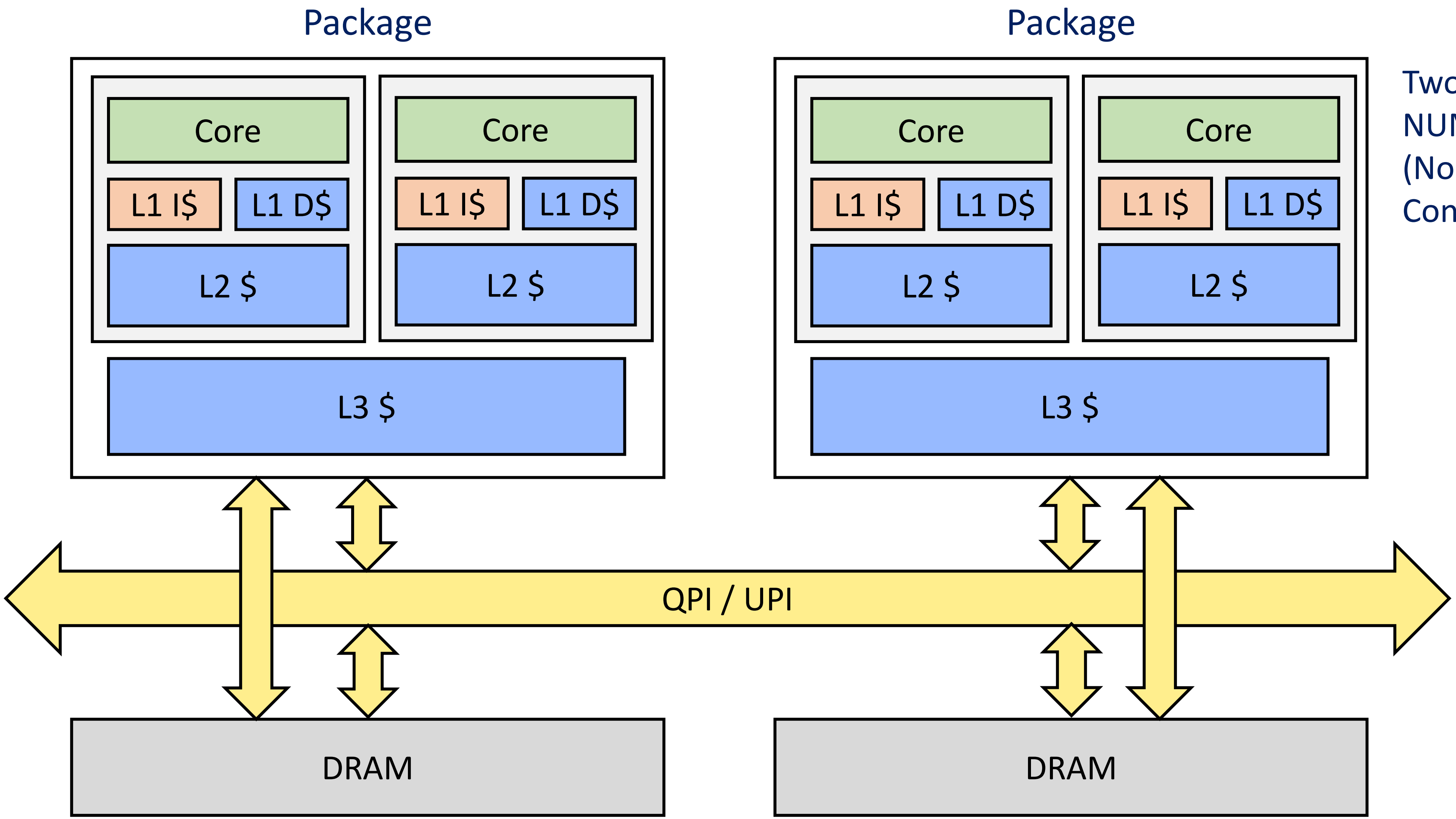


Real-world: Intel Haswell i7

- Cache access latencies
  - L1: 4 - 5 cycles
  - L2: 12 cycles
  - L3: ~30 - ~50 cycles
- For reference, Haswell as 14 pipeline stages

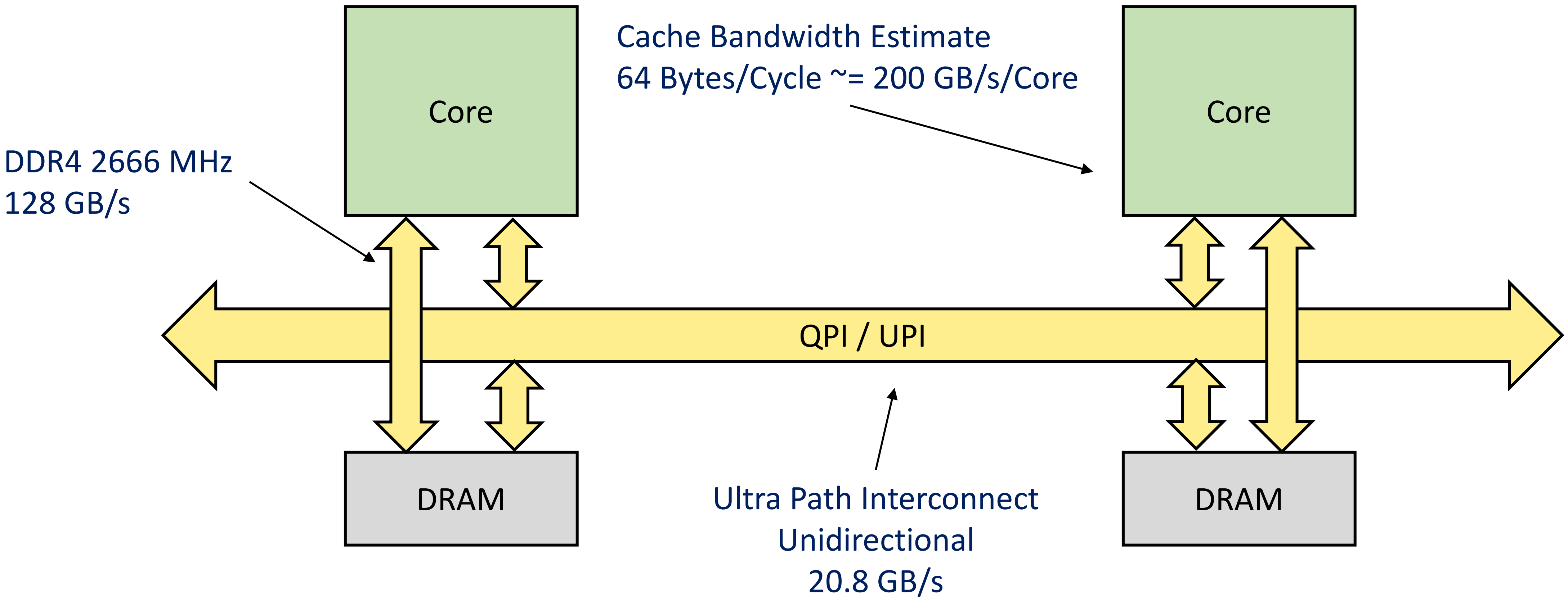
As soon as we miss L1 cache, there is performance overhead!

Multi-Core Memory System Architecture



Two packages make up a NUMA (Non-Uniform Memory Access) Configuration

Memory System Bandwidth Snapshot



Memory/PCIe controller used to be on a separate “North bridge” chip, now integrated on-die
All sorts of things are now on-die! Even network controllers! (Specialization!)

Reminder: Cache Coherency

- Cache coherency
  - Informally: Read to **each address** must return the most recent value
  - Typically: All writes must be visible at some point, and in proper order
- Coherency protocol implemented between each core's private caches
  - MSI, MESI, MESIF, …
  - Won't go into details here
- Simply put:
  - When a core writes a cache line
  - All other instances of that cache line needs to be invalidated
- Emphasis on ***cache line***

Cache Prefetching

- CPU speculatively prefetches cache lines
  - While CPU is working on the loaded 64 bytes, 64 more bytes are being loaded
- Hardware prefetcher is usually not very complex/smart
  - Sequential prefetching (N lines forward or backwards)
  - Strided prefetching
- Programmer-provided prefetch hints
  - __builtin_prefetch(address, r/w, temporal locality?); for GCC
  - Will generate prefetch instructions if available on architecture

Now That’s Out of The Way…

CS250P: Computer Systems Architecture
Performance Engineering with Caches

Sang-Woo Jun

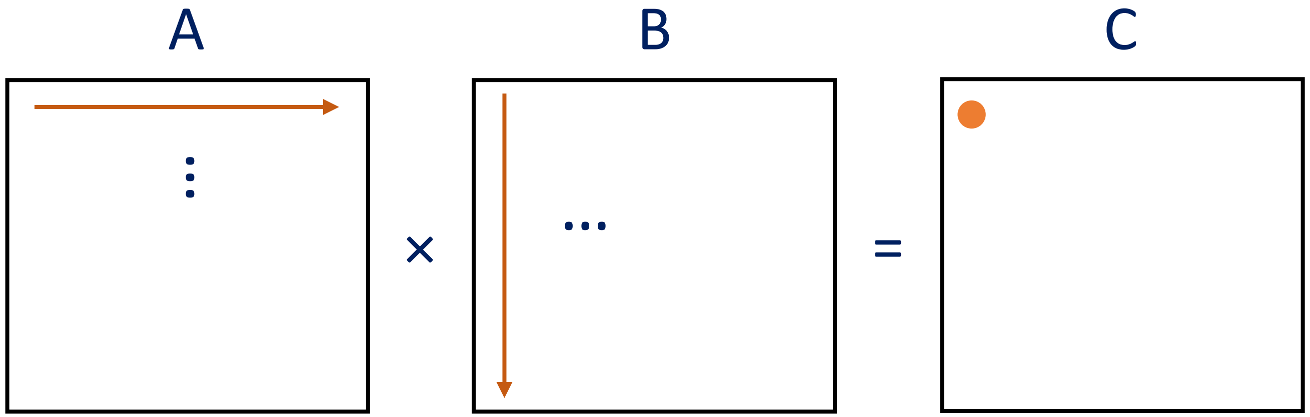
Fall 2022

UCI

Cache Efficiency Issue #1: Cache Line Size
Matrix Multiplication and Caches

- Multiplying two NxN matrices (C = A × B)

```
for (i = 0 to N)
   for (j = 0 to N)
      for (k = 0 to N)
         C[i][j] += A[i][k] * B[k][j]
```



2048*2048 on a i5-7400 @ 3 GHz using GCC –O3 = 63.19 seconds

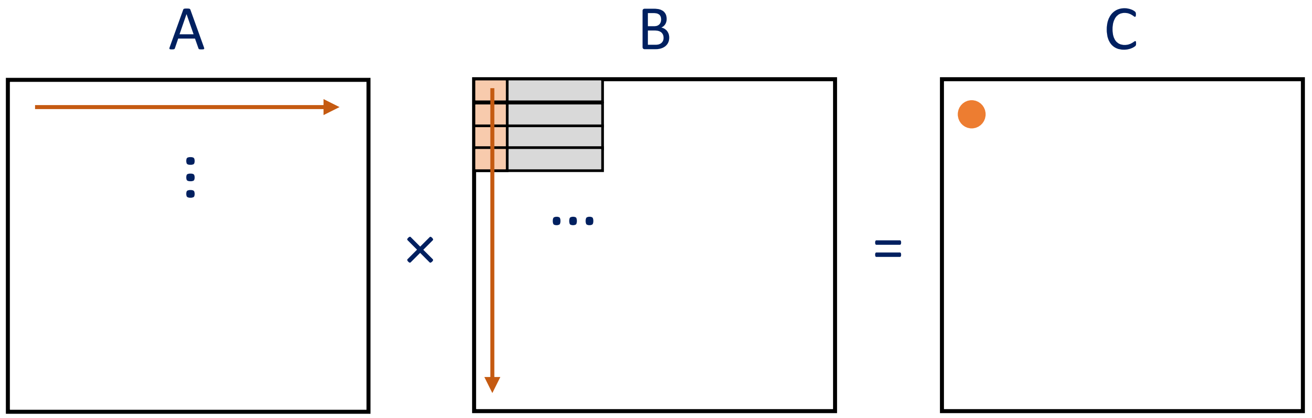
is this fast?

Whole calculation requires 2K * 2K * 2K = 8 Billion floating-point mult + add
At 3 GHz, ~5 seconds just for the math. Over 1000% overhead!

Cache Efficiency Issue #1: Cache Line Size Matrix Multiplication and Caches

- Column-major access makes inefficient use of cache lines
  - A 64 Byte block is read for each element loaded from B
  - 64 bytes read from memory for each 4 useful bytes
- Shouldn't caching fix this? Unused bits should be useful soon!
  - 64 bytes x 2048 = 128 KB … Already overflows L1 cache (~32 KB)

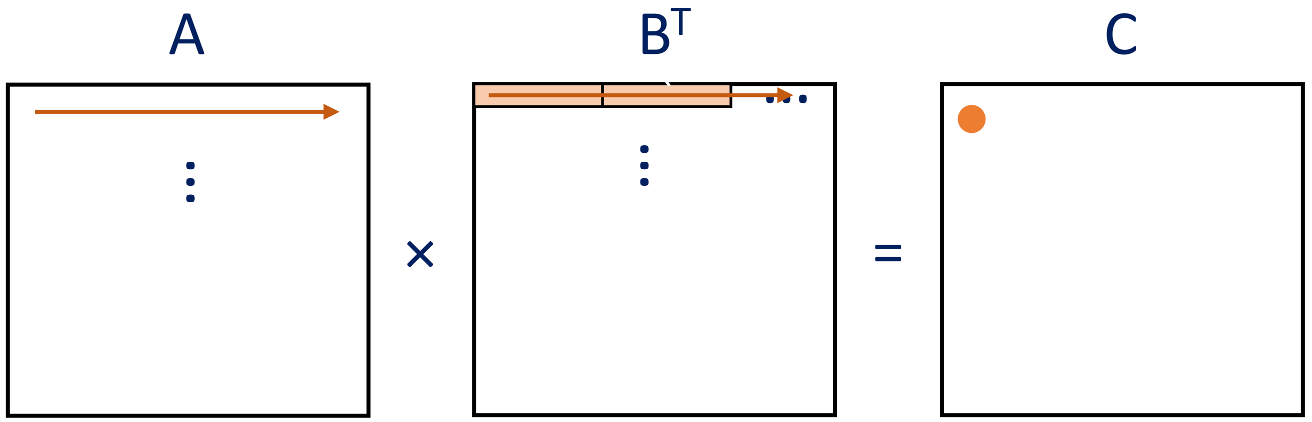
```
for (i = 0 to N)
  for (j = 0 to N)
    for (k = 0 to N)
      C[i][j] += A[i][k] * B[k][j]
```



Cache Efficiency Issue #1: Cache Line Size
Matrix Multiplication and Caches

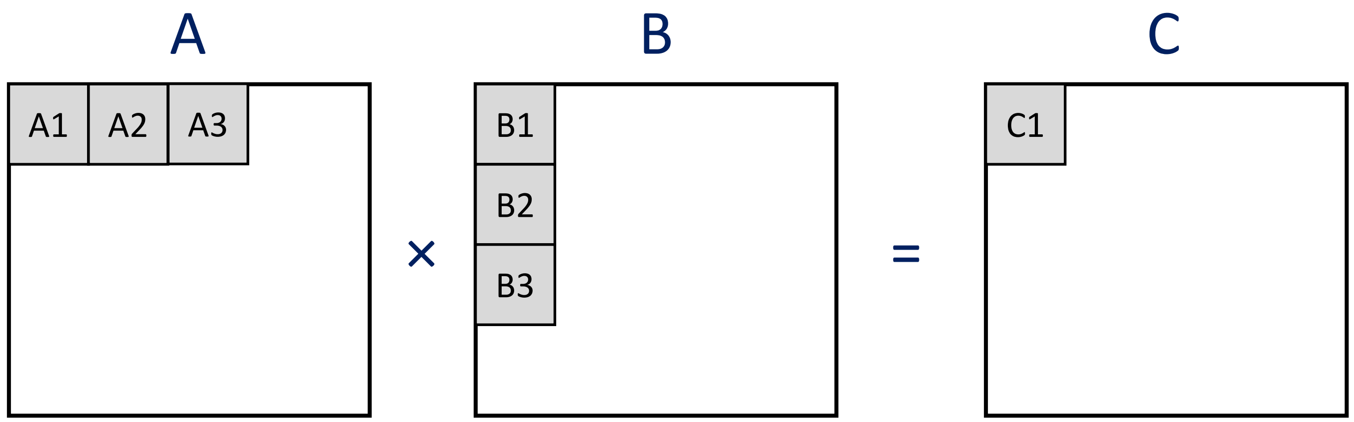
- One solution: Transpose B to match cache line orientation
  - Does transpose add overhead? Not very much as it only scans B once
- Drastic improvements!
  - Before: 63.19s
  - After: 10.39s … 6x improvement!
  - But still not quite ~5s

```
for (i = 0 to N)
  for (j = 0 to N)
    for (k = 0 to N)
      C[i][j] += A[i][k] * Bt[j][k]
```



Cache Efficiency Issue #2: Capacity Considerations

- Performance is best when working set fits into cache
  - But as shown, even 2048 x 2048 doesn't fit in cache
  - -> 2048 * 2048 * 2048 elements read from memory for matrix B
- Solution: Divide and conquer! – Blocked matrix multiply
  - For block size 32 × 32 -> 2048 * 2048 * (2048/32) reads



C1 sub-matrix = A1×B1 + A2×B2 + A3×B3 …

Blocked Matrix Multiply Evaluations

Benchmark	Elapsed (s)	Normalized Performance
Naïve	63.19	1
Transposed	10.39	6.08
Blocked Transposed	7.35	8.60

- Blocked Transposed bottlenecked by computation
  - Peak theoretical FLOPS for my processor running at 3 GHz ~= 3 GFLOPS
  - 7.35s for matrix multiplication ~= 2.18 GFLOPS
  - Not bad, considering need for branches and other instructions!
  - L1 cache access now optimized, but not considers larger caches

Blocked Matrix Multiply Evaluations

Benchmark	Elapsed (s)	Normalized Performance
Naïve	63.19	1
Transposed	10.39	6.08
Blocked (32)	7.35	8.60

Bottlenecked by computation

Bottlenecked by memory

Bottlenecked by processor

Bottlenecked by memory (Not scaling!)

- AVX Transposed reading from DRAM at 14.55 GB/s
  - 2048^3 * 4 (Bytes) / 2.20 (s) = 14.55 GB/s
  - 1x DDR4 2400 MHz on machine -> 18.75 GB/s peak
  - Pretty close! Considering DRAM also used for other things (OS, etc)
- Multithreaded getting 32 GB/s effective bandwidth
  - Cache effects with small chunks

Aside: Cache oblivious algorithms

- For sub-block size B × B -> N * N * (N/B) reads. What B do we use?
  - Optimized for L1? (32 KiB for me, who knows for who else?)
  - If B*B exceeds cache, sharp drop in performance
  - If B*B is too small, gradual loss of performance
- Do we ignore the rest of the cache hierarchy?
  - Say B optimized for L3,
B × B multiplication is further divided into T×T blocks for L2 cache
  - T × T multiplication is further divided into U×U blocks for L1 cache
  - … If we don't, we lose performance
- Class of "cache-oblivious algorithms"

Typically recursive definition of data structures… topic for another day

Aside: Recursive Matrix Multiplication

$$\underbrace{\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}}_{C} = \underbrace{\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}}_{B}$$

$$= \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$$

8 multiply-adds of $(n/2) \times (n/2)$ matrices
Recurse down until very small

Blocked Matrix Multiply Evaluations

Benchmark	Elapsed (s)	Normalized Performance
Naïve	63.19	1
Transposed	10.39	6.08
Blocked (32)	7.35	8.60
AVX Transposed	2.20	28.72
Blocked (32) AVX	1.50	42.13
4 Thread Blocked (32) AVX	1.09	57.97

- Using FMA SIMD, Cache-Oblivious AVX gets 19 GFLOPS
  - Theoretical peak is 3 GHz x 8 way SIMD == 24 GFLOPS… Close!

140x performance increase compared to the baseline!

Writing Cache Line Friendly Software

- (Whenever possible) use data in coarser-granularities
  - Each access may load 64 bytes into cache, make use of them!
  - e.g., Transposed matrix B in matrix multiply, blocked matrix multiply
- Many profilers will consider the CPU “busy” when waiting for cache
  - Can’t always trust “CPU utilization: 100%”

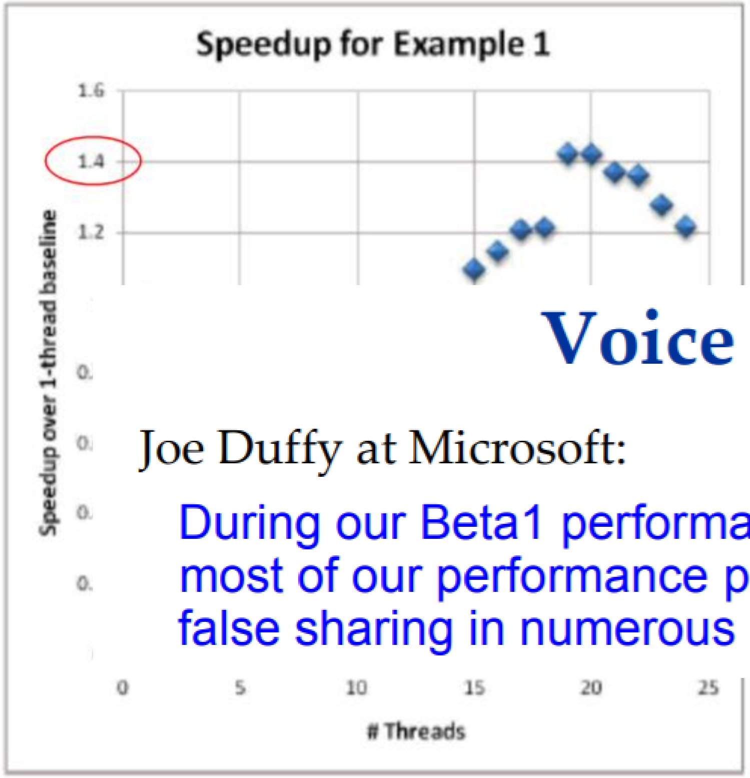
Aside: Object-Oriented Programming And Caches

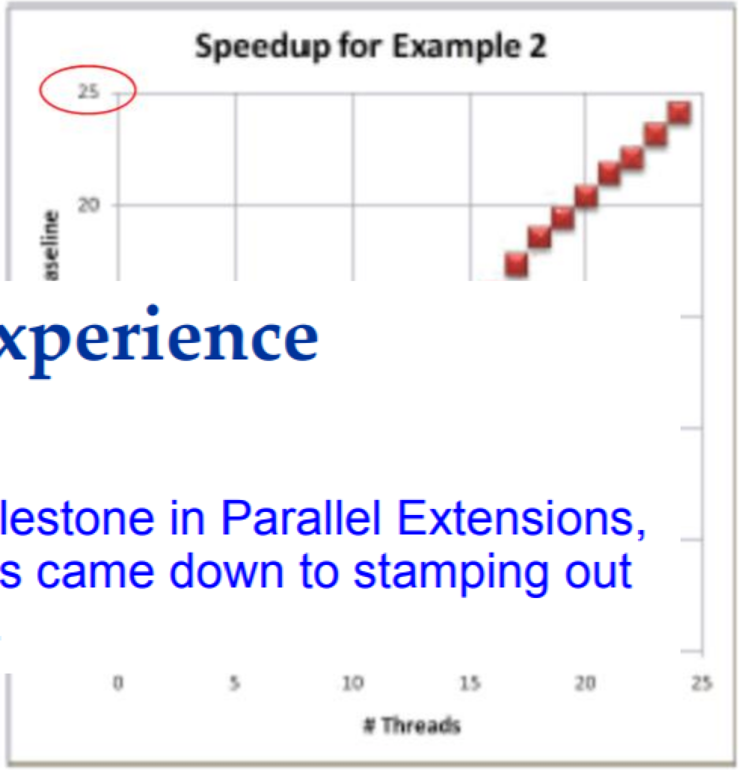
- OOP wants to collocate all data for an entity in a class/struct
  - All instance variables are located together in memory
- Cache friendly OOP
  - All instance variables are accessed whenever an instance is accessed
- Cache unfriendly OOP
  - Only a small subset of instance variables are accessed per instance access
  - e.g., a “for” loop checking the “valid” field of all entities
    - 1 byte accessed per cache line read!
- Non-OOP solution: Have a separate array for “valid”s
  - Is this a desirable solution? Maybe…

Cache Efficiency Issue #3: False Sharing

- Different memory locations, written to by different cores, mapped to same cache line
  - Core 1 performing “results[0]++;”
  - Core 2 performing “results[1]++;”
- Remember cache coherence
  - Every time a cache is written to, all other instances need to be invalidated!
  - “results” variable is ping-ponged across cache coherence every time
  - Bad when it happens on-chip, terrible over processor interconnect (QPI/UPI)
- Simple solution: Store often-written data in local variables

Removing False Sharing





Voice of Experience

Joe Duffy at Microsoft:

During our Beta1 performance milestone in Parallel Extensions, most of our performance problems came down to stamping out false sharing in numerous places.

With False Sharing

Without False Sharing

Aside: Non Cache-Related Optimizations: Loop Unrolling

- Increase the amount of work per loop iteration
  - Improves the ratio between computation instructions and branch instructions
  - Compiler can be instructed to automatically unroll loops
  - Increases binary size, because unrolled iterations are now duplicated code

<table>
<tr><th>Normal loop</th><th>After loop unrolling</th></tr>
<tr><td><pre>int x;
for (x = 0; x < 100; x++)
{
    delete(x);
}</pre></td><td><pre>int x;
for (x = 0; x < 100; x += 5)
{
    delete(x);
    delete(x + 1);
    delete(x + 2);
    delete(x + 3);
    delete(x + 4);
}</pre></td></tr>
</table>

Source: Wikipedia “Loop unrolling”

Aside: Non Cache-Related Optimizations: Function Inlining

- A small function called very often may be bottlenecked by call overhead
- Compiler copies the instructions of a function into the caller
  - Removes expensive function call overhead (stack management, etc)
  - Function can be defined with "inline" flag to hint the compiler
    - "inline int foo()", instead of "int foo()"
- Personal anecdote
  - Inlining a key (very small) kernel function resulted in a 4x performance boost

Issue #4
Instruction Cache Effects

- Instructions are also stored in cache
  - L1 cache typically has separate instances for instruction and data caches
    - In most x86 architectures, 32 KiB each
    - L2 onwards are shared
  - Lots of spatial locality, so miss rate is usually very low
    - On SPEC, ~2% at L1
  - But adversarial examples can still thrash the cache
- Instruction cache often has dedicated prefetcher
  - Understands concepts of branches and function calls
  - Prefetches blocks of instructions without branches

Optimizing Instruction Cache

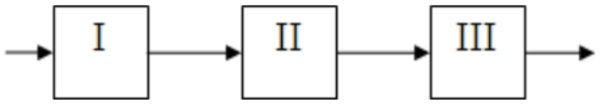
- Instruction cache misses can effect performance
  - "Linux was routing packets at ~30Mbps [wired], and wireless at ~20. Windows CE was crawling at barely 12Mbps wired and 6Mbps wireless.
  - [...] After we changed the routing algorithm to be more cache-local, we started doing 35Mbps [wired], and 25Mbps wireless – 20% better than Linux.
    – Sergey Solyanik, Microsoft
  - [By organizing function calls in a cache-friendly way, we] achieved a 34% reduction in instruction cache misses and a 5% improvement in overall performance.
    -- Mircea Livadariu and Amir Kleen, Freescale

Improving Instruction Cache Locality #1

- Careful with loop unrolling
  - They reduce branching overhead, but reduces effective I$ size
  - When gcc's –O3 performs slower than –O2, this is usually what's happening
- Careful with function inlining
  - Inlining is typically good for very small* functions
  - A rarely executed path will just consume cache space if inlined
- Move conditionals to front as much as possible
  - Long paths of no branches good fit with instruction cache/prefetcher

Improving Instruction Cache Locality #2

- Organize function calls to create temporal locality




```
for (i=0;i<N;i++)
{
        temp=stage_I(input[i]);
        temp=stage_II(temp);
        output[i]= stage_III(temp);
}
```

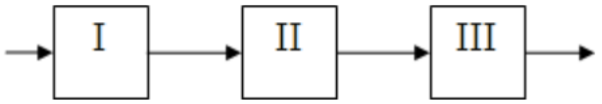
Baseline: Sequential algorithm

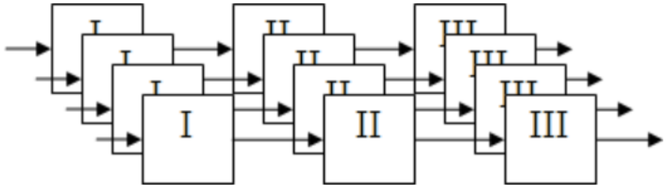
If the functions stage_I, stage_II, and stage_III are sufficiently large, their instructions will thrash the instruction cache!

Livadariu et. al., “Optimizing for instruction caches,” EETimes

Improving Instruction Cache Locality #2

- Organize function calls to create temporal locality






```
for (i=0;i<N;i++)
{
        temp=stage_I(input[i]);
        temp=stage_II(temp);
        output[i]= stage_III(temp);
}
```

Baseline: Sequential algorithm

```
for (i=0;i<N;i++)
        temp[i]=stage_I(input[i]);
for (i=0;i<N;i++)
        temp[i]=stage_II(temp[i]);
for (i=0;i<N;i++)
        output[i]= stage_III(temp[i]);
```

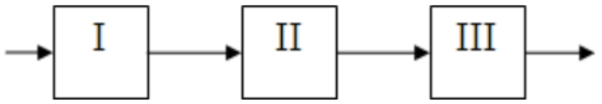
Ordering changed for cache locality

New array “temp” takes up space. N could be large!

Livadariu et. al., “Optimizing for instruction caches,” EETimes

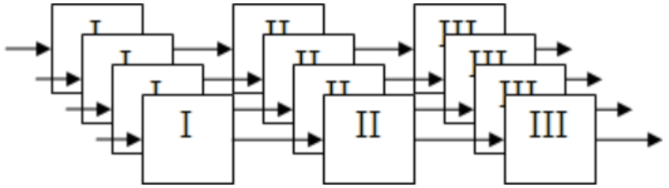
Improving Instruction Cache Locality #2

- Organize function calls to create temporal locality




```
for (i=0;i<N;i++)
{
        temp=stage_I(input[i]);
        temp=stage_II(temp);
        output[i]= stage_III(temp);
}
```

Baseline: Sequential algorithm




```
for (i=0;i<N;i++)
        temp[i]=stage_I(input[i]);
for (i=0;i<N;i++)
        temp[i]=stage_II(temp[i]);
for (i=0;i<N;i++)
        output[i]= stage_III(temp[i]);
```

Ordering changed for cache locality

```
for (j=0;j<N;j+=M)
{
        for (i=0;i<M;i++)
                temp[i]=stage_I(input[j+i]);
        for (i=0;i<M;i++)
                temp[i]=stage_II(temp[j+i]);
        for (i=0;i<M;i++)
                output[i]= stage_III(temp[j+i]);
}
```

Balance to reduce memory footprint

Livadariu et. al., “Optimizing for instruction caches,” EETimes