CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 32: Caches Part 4

Instructors: Dan Garcia, Justin Yokota

Computing in the News

Simple algorithm brings complex power to quantum computers

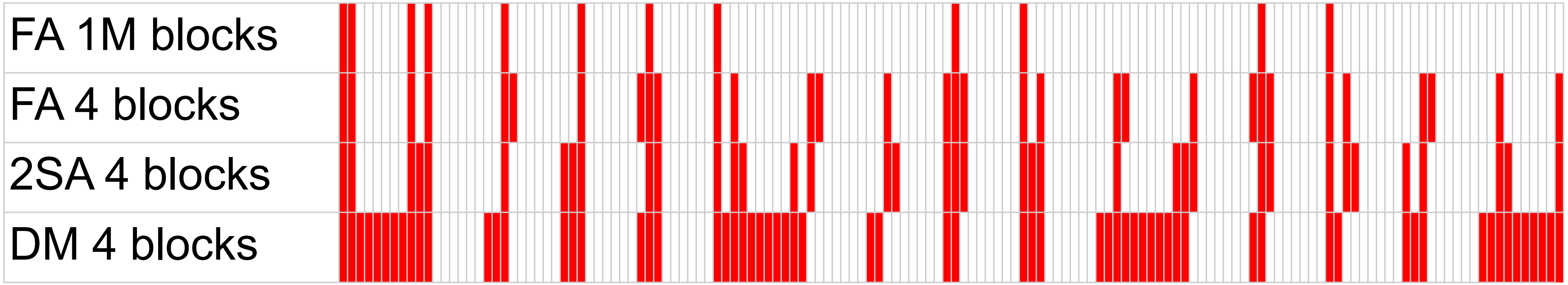
A novel protocol for quantum computers could reproduce the complex dynamics of quantum materials

The prototype quantum computers demonstrated to date have achieved time-evolution operators using a relatively simple technique called Trotterization. But Trotterization is thought to be unsuitable for the quantum computers of the future because it requires a huge number of quantum gates and thus a lot of computational time. Consequently, researchers have been striving to create quantum algorithms for accurate quantum simulations that use fewer quantum gates.

Now, Mizuta, working with colleagues from across Japan, has proposed a much more efficient and practical algorithm. A hybrid of quantum and classical methods, it can compile time-evolution operators at a lower computational cost, enabling it to be executed on small quantum computers, or even conventional ones.

Analyzing Cache Effectiveness

- Let's compare the hit patterns of a few cache types
- The below diagram shows the hit/miss pattern of various caches when run on the Matrix Multiply example shown in lecture
  - Red = Miss, White = Hit
- Also one more cache: A fully associative cache with an absurd number of blocks (an "ideal" cache which will hit as often as possible)



Analyzing Cache Effectiveness

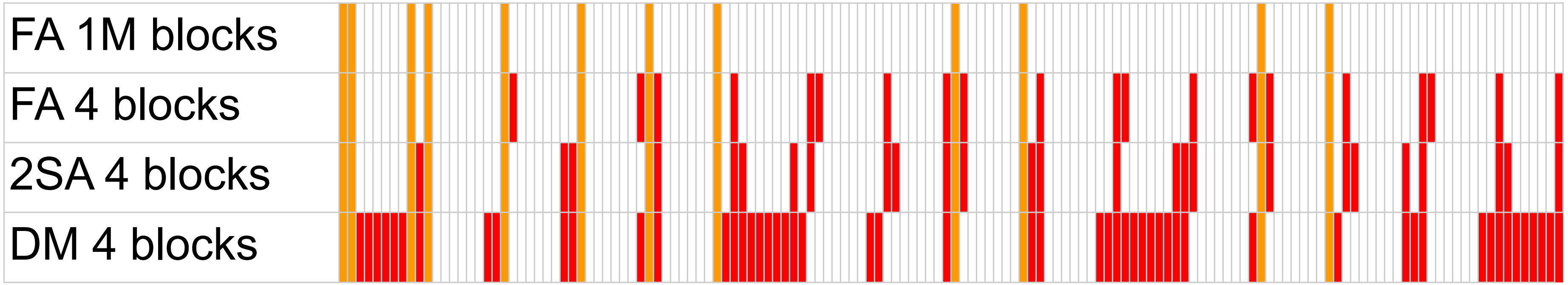
- When looking at cache hit/miss patterns, there were several interesting patterns
  - When we accessed a new set of cache blocks, there were significantly more misses than usual
  - When associativity increased, many misses were converted into hits, though a few hits turned into misses
- If we wanted to improve our cache structure, we'd need to know why misses happened
  - Do we increase associativity (and maybe decrease size to keep a constant hit time), or increase cache size (and maybe decrease associativity to keep a constant hit rate)?
- Goal of today: Classify misses according to why they occurred, so that we can figure out how we might change our cache to improve hit rate
  - Program-independent analysis, so we'll restrict to statements that can be made just off the cache state.

Miss Classifications: Definitional Goals

- Ideally, we have three main types of misses:
    - Edge cases cause this ideal to be impossible, but our final definitions will match this as best as possible
- Compulsory Misses: Misses that are impossible to avoid; regardless of how we set up our cache
- Capacity Misses: Misses that were the result of our cache having limited space (not enough capacity)
- Conflict Misses: Misses that were added as a result of a low associativity/thrashing (blocks conflicted)

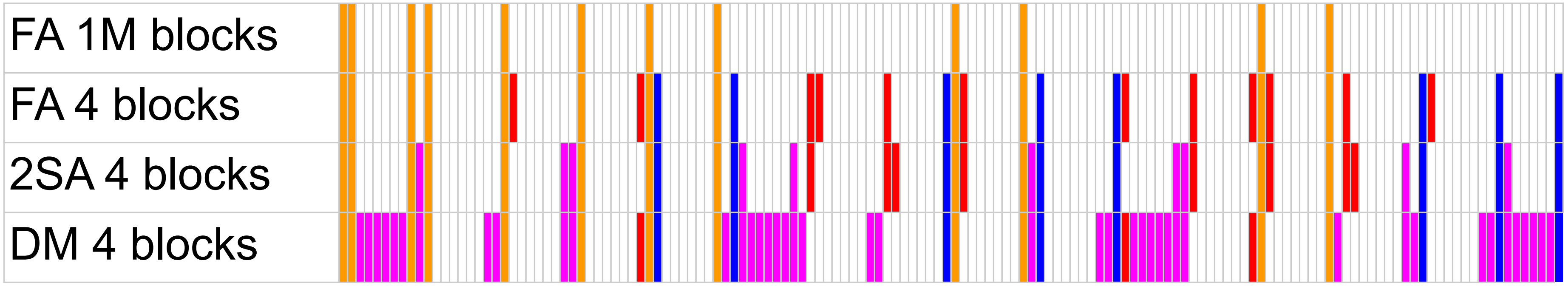
Compulsory Miss

- Misses that are impossible to avoid; regardless of how we set up our cache
- All of these are the first access to a block
  - Ex. Accessing A[0] for the first time
- Guaranteed to be misses, so there's no real way to prevent these (beyond increasing block size).
  - If most misses are compulsory, we can't improve hit rate much.



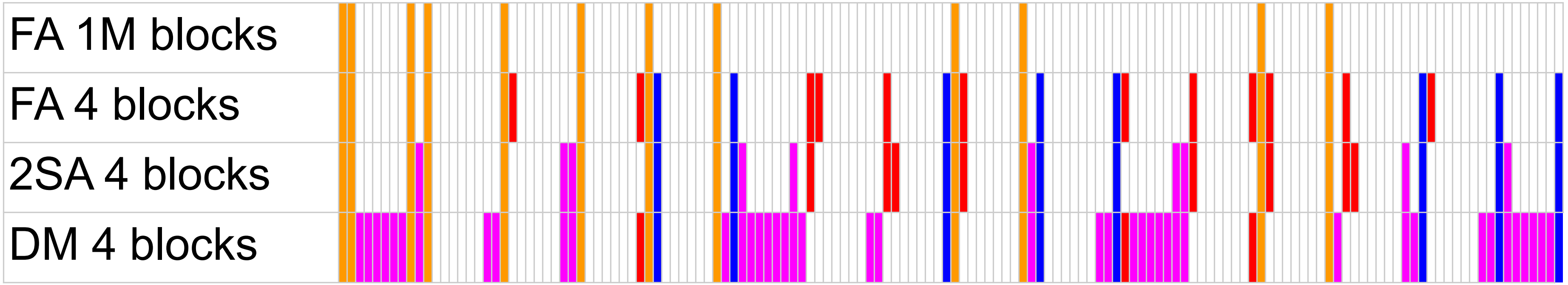
Conflict Miss

- Misses that were added as a result of a low associativity/thrashing (blocks conflicted)
- Intended to be misses that were added by decreased associativity
- https://bitsavers.org/pdf/dec/tech_reports/WRL-TN-53.pdf defines them as "misses that would not occur if the cache was fully-associative and had LRU replacement" (with all other noncompulsory misses being capacity)



Conflict Miss

- The problem with this definition is that it assumes that all misses in the Fully Associative cache also cause misses in lower associativities
    - This isn't actually true; as we see below, sometimes we get "conflict hits" because a block that would have been evicted ends up staying in the cache longer
    - Also requires a lot more work to compute, so we'll use an easier definition in this class



Agenda

- Miss Classifications
- Multilevel Caches
- Caches in Multithreaded Systems
- MOESI

Miss Classifications: Definitions in this class

- Compulsory Misses: Misses that would occur even if the cache was a fully associative cache with infinite space
- Capacity Misses: Misses that are neither conflict nor compulsory
- Conflict Misses: Misses that would not have happened if the cache was fully associative under at least one consistent eviction policy
- Consistent eviction policy: An eviction policy that keeps all the data in the lower-associativity cache
  - In other words, any eviction policy such that the data in the Fully Associative cache is a superset of the data in our actual cache.
  - This definition guarantees that we don't have "conflict hits", but also tends to classify misses more as capacity misses than as conflict misses

Miss Classifications: Simplified Equivalent Definitions

- If a miss occurs, look at the most recent time that that block was evicted
- If that block was never evicted before, it's a compulsory miss
- If the block was evicted when the cache was full (no empty spaces), it's a capacity miss
- If the block was evicted when the cache was not full (at least one empty space), it's a conflict miss
- Let's look again at the 2-way set associative cache example

Caching Example: Hit Classifications

- Start: Cache starts cold
  - We'll restrict the cache to only the parts that are important (no data/dirty/etc, block instead of tag).
  - Also will write the status of each block on the right
  - Blocks will be ordered by LRU within a set (unlike an actual cache)
- First miss: A[0]
  - Compulsory miss

Index	Block	Index	Block
0	–	1	–
0	–	1	–

Block	Status	Block	Status
A[0]	NA	B[8]	NA
A[4]	NA	B[12]	NA
A[8]	NA	C[0]	NA
A[12]	NA	C[4]	NA
B[0]	NA	C[8]	NA
B[4]	NA	C[12]	NA

Caching Example: Hit Classifications

- Next miss is B[0]
  - Compulsory Miss
- Next miss is C[0]
  - Compulsory Miss, evicts A[0]
  - A[0] was evicted even though there were empty slots, so the next access of A[0] will be a conflict miss

Block	Status	Block	Status
A[0]	In Cache	B[8]	NA
A[4]	NA	B[12]	NA
A[8]	NA	C[0]	NA
A[12]	NA	C[4]	NA
B[0]	NA	C[8]	NA
B[4]	NA	C[12]	NA

Index	Block	Index	Block
0	A[0]	1	-
0	-	1	-

Caching Example: Hit Classifications

- A[0]
  - Conflict miss
  - Evicts B[0] (conflict)
- B[4]
  - Compulsory miss
- B[8]
  - Compulsory miss
  - Evicts C[0] (conflict)

Index	Block	Index	Block
0	`B[0]`	1	–
0	`C[0]`	1	–

Block	Status	Block	Status
A[0]	Conflict	B[8]	NA
A[4]	NA	B[12]	NA
A[8]	NA	C[0]	In Cache
A[12]	NA	C[4]	NA
B[0]	In Cache	C[8]	NA
B[4]	NA	C[12]	NA

Caching Example: Hit Classifications

- C[2]
  - Conflict miss
  - Evicts A[0] (conflict)
- A[0]
  - Conflict miss
  - Evicts B[8] (conflict)
- B[12]
  - Compulsory miss
  - At this point, all blocks are used, so any further evictions will cause capacity misses

Block	Status	Block	Status
A[0]	In Cache	B[8]	In Cache
A[4]	NA	B[12]	NA
A[8]	NA	C[0]	Conflict
A[12]	NA	C[4]	NA
B[0]	Conflict	C[8]	NA
B[4]	In Cache	C[12]	NA

Index	Block	Index	Block
0	`A[0]`	1	`B[4]`
0	`B[8]`	1	-

Caching Example: Hit Classifications

- A[4]
  - Compulsory miss
  - Evicts B[4] (will cause a capacity miss next)
- B[0]
  - Conflict miss
  - Evicts C[0] (capacity)
- C[4]
  - Compulsory miss
  - Evicts B[12] (capacity)

Block	Status	Block	Status
A[0]	In Cache	B[8]	Conflict
A[4]	NA	B[12]	In Cache
A[8]	NA	C[0]	In Cache
A[12]	NA	C[4]	NA
B[0]	Conflict	C[8]	NA
B[4]	In Cache	C[12]	NA

Index	Block	Index	Block
0	C[0]	1	B[4]
0	A[0]	1	B[12]

Caching Example: Hit Classifications

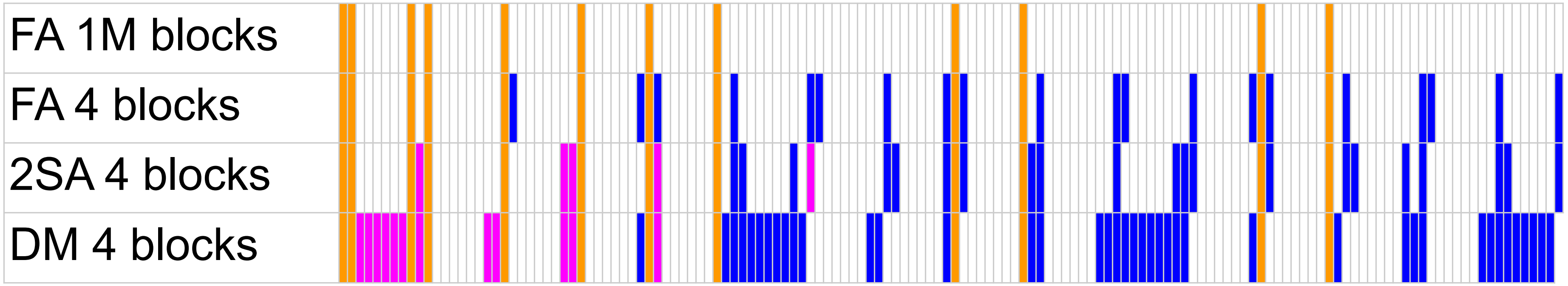
- B[4]
  - Capacity miss
  - Evicts A[4] (capacity)
- A[5]
  - Capacity miss
  - Evicts C[4] (capacity)
- C[5]
  - Capacity miss
  - Evicts B[4] (capacity)
  - This is the downside of this definition (this really feels like it should be a conflict miss)

Block	Status	Block	Status
A[0]	In Cache	B[8]	Conflict
A[4]	In Cache	B[12]	Capacity
A[8]	NA	C[0]	Capacity
A[12]	NA	C[4]	In Cache
B[0]	In Cache	C[8]	NA
B[4]	Capacity	C[12]	NA

Index	Block	Index	Block
0	`A[0]`	1	`A[4]`
0	`B[0]`	1	`C[4]`

Overall Miss Classifications

- As noted, this definition tends to classify things as capacity more often than it feels like it should
- Still, it works well enough for the initial few accesses, and is simpler to implement, so we'll follow this definition for this class, just for simplicity



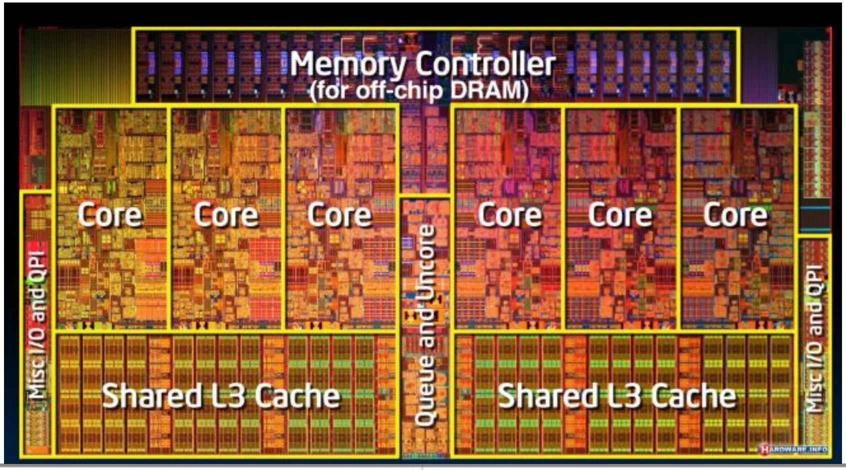
How to Improve your Cache Hit Rate

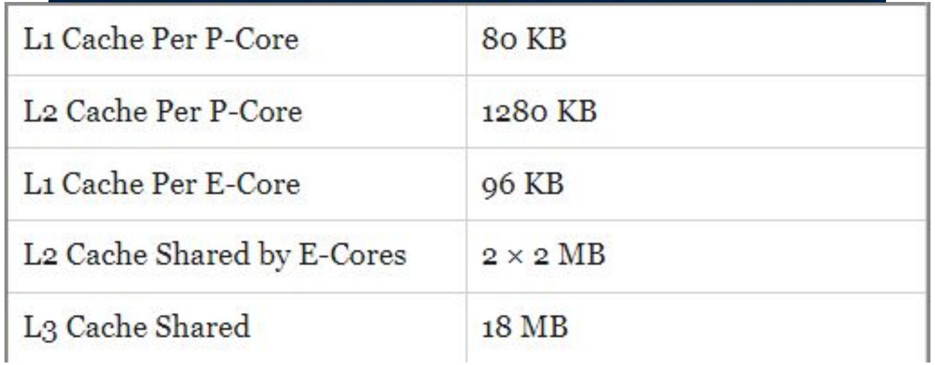
- Look at what type of miss is most common
  - Different access patterns will have different miss patterns, so test some "canonical" access sequence, like matrix multiplication
- If Compulsory misses are most common
  - Not really much you can do. Focus on something else to optimize
- If Capacity misses are most common
  - You're using too much memory at once for your cache to be efficient.
  - From the hardware perspective, increase cache size
  - From the software perspective, reduce the "working block"; try to split your code into parts that access only as much memory as your cache can store
- If Conflict misses are most common
  - Your cache needs more associativity
  - From the hardware side, increase associativity
  - From the software side, reduce the number of distinct chunks of memory you're accessing (though conflict misses tend to be minimal in most real-life caches)

Multilevel Caches

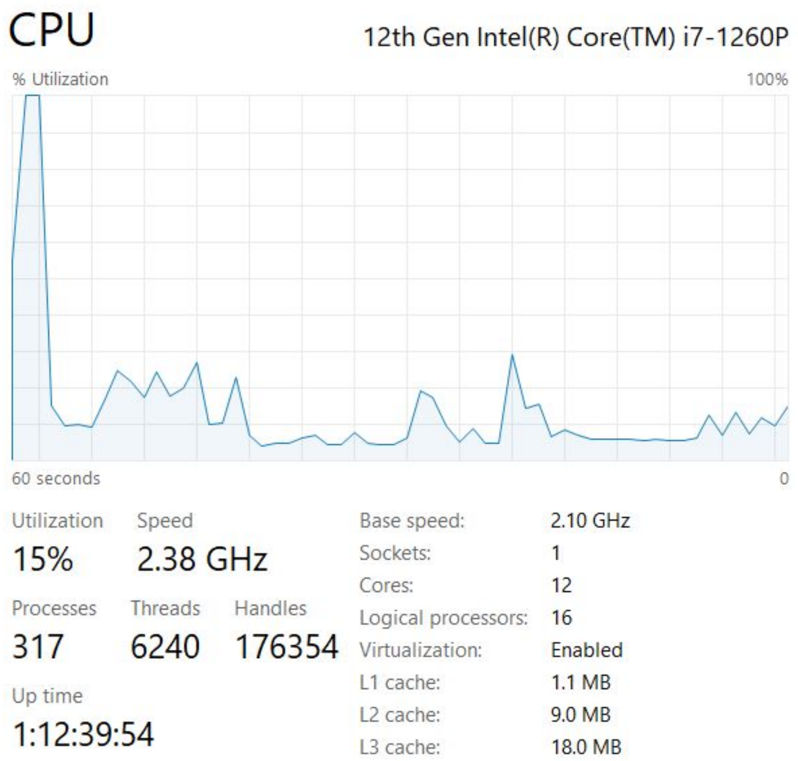
- Current setup: If cache hit, then fast access. If cache miss, then slow access
- Problem: Hit rate scales with access time; the higher the hit rate we want, the slower we'll end up making cache accesses. With just one cache, we need to make some tough trade-offs
- Solution: Add multiple layers of caches.
    - Check L1 cache. If hit, we're done
    - If miss, check L2 cache. If hit, we're done, and bring data up to L1 cache
    - If miss, check L3 cache. If hit, we're done, and bring data up to L1 and L2 cache
    - …
- Each level of cache is larger than the previous and should have a subset of the data in the next cache
- Library analogy: We buy a bookshelf for right next to us, then we buy a larger bookshelf to put in our garage, then make a shed to store more books.

Multilevel Caches





L1 Cache Per P-Core	80 KB
L2 Cache Per P-Core	1280 KB
L1 Cache Per E-Core	96 KB
L2 Cache Shared by E-Cores	2 × 2 MB
L3 Cache Shared	18 MB



CPU 12th Gen Intel(R) Core(TM) i7-1260P

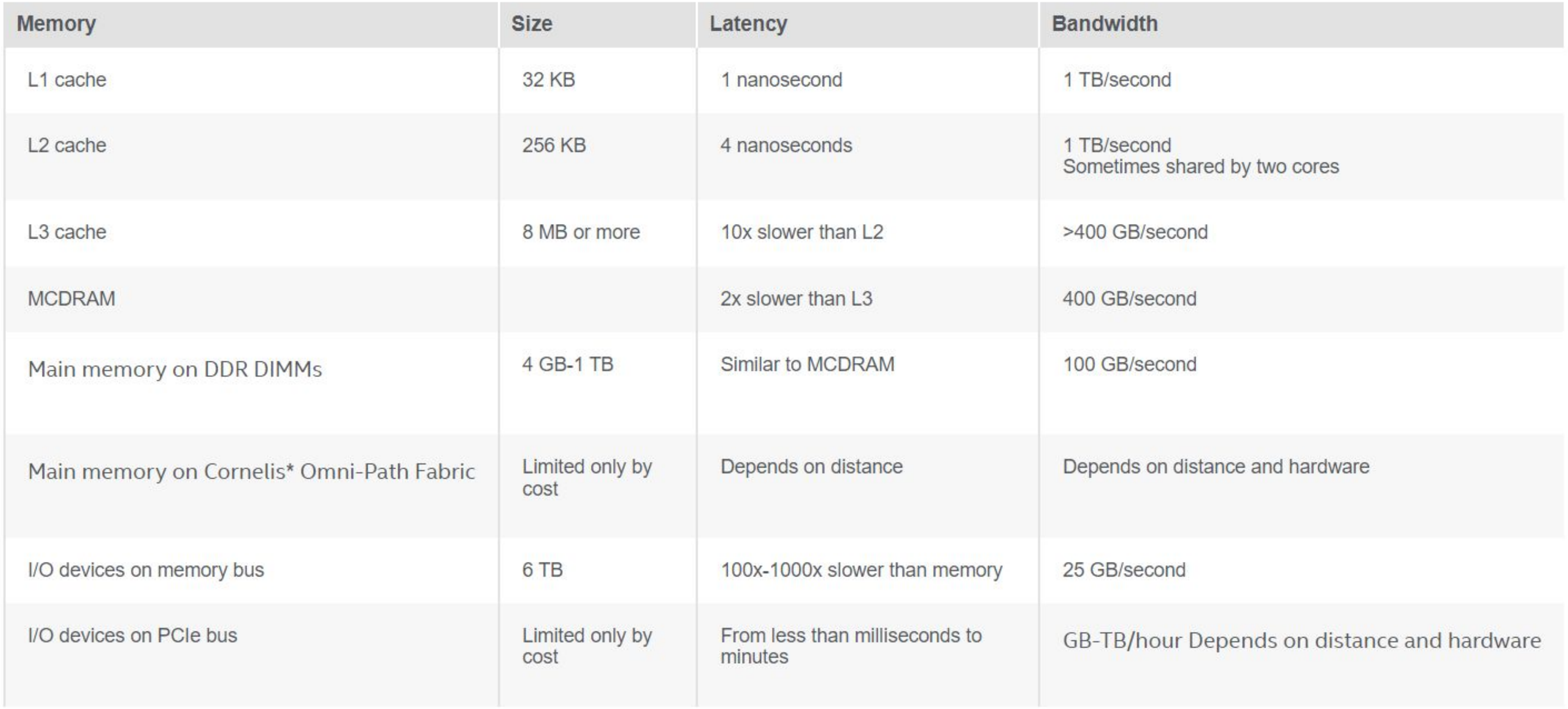
% Utilization 100%

60 seconds 0

Utilization	Speed	
15%	2.38 GHz	
Processes	Threads	Handles
317	6240	176354
Up time		
1:12:39:54		

Base speed:	2.10 GHz
Sockets:	1
Cores:	12
Logical processors:	16
Virtualization:	Enabled
L1 cache:	1.1 MB
L2 cache:	9.0 MB
L3 cache:	18.0 MB

Multilevel Caches



Memory	Size	Latency	Bandwidth
L1 cache	32 KB	1 nanosecond	1 TB/second
L2 cache	256 KB	4 nanoseconds	1 TB/second Sometimes shared by two cores
L3 cache	8 MB or more	10x slower than L2	>400 GB/second
MCDRAM		2x slower than L3	400 GB/second
Main memory on DDR DIMMs	4 GB-1 TB	Similar to MCDRAM	100 GB/second
Main memory on Cornelis* Omni-Path Fabric	Limited only by cost	Depends on distance	Depends on distance and hardware
I/O devices on memory bus	6 TB	100x-1000x slower than memory	25 GB/second
I/O devices on PCIe bus	Limited only by cost	From less than milliseconds to minutes	GB-TB/hour Depends on distance and hardware

Multilevel Caches

- Generally, the L1 cache is small, but also on the core
  - Each core gets its own L1 cache
  - Fast access, but also fairly small
- L2 cache is larger and slightly slower, but usually a bit farther away
  - Sometimes shared, sometimes on the core
- L3 cache is normally much larger and much slower, and is shared with all cores
- Using the numbers from the previous page as examples:
  - L1 hit time = 1 ns
  - L2 hit time = 1 ns + 4 ns = 5 ns
  - L3 hit time = 1 ns + 4 ns + 40 ns = 45 ns
  - L3 miss = 1 ns + 4 ns + 40 ns + 80 ns = 125 ns
- Miss rate is calculated only for accesses that "reach" that cache
  - L2 hit rate is % of L2 accesses that hit, so L1 hits don't count as L2 hits OR L2 misses

AMAT in Multilevel Caches

- Let's say we have a memory system with the following properties. What would be the AMAT of this system?
- Respond at PollEv.com/jy314

	Hit time	Hit rate
L1 Cache	1 ns	50%
L2 Cache	4 ns	75%
L3 Cache	40 ns	80%
SRAM	80 ns	100%

What is the AMAT of the cache in the previous slide?

Top

AMAT in Multilevel Caches

- Let's say we have a memory system with the following properties. What would be the AMAT of this system?
- 50% of accesses are L1 hits
- 50% of accesses are L1 misses
  - 50%*75% = ⅜ are L2 hits
- ⅛ of accesses are L2 misses
  - ⅛*80% = 1/10 are L3 hits
- 1/40 of accesses are L3 misses
  - And therefore access SRAM
- Total: 1 ns + 0.5*4 ns + ⅛ * 40 ns + 1/40 * 80 ns
- 10 ns access time = 8x speedup

	Hit time	Hit rate
L1 Cache	1 ns	50%
L2 Cache	4 ns	75%
L3 Cache	40 ns	80%
SRAM	80 ns	100%

Caching with multithreading

- Each core has its own L1 cache, and sometimes its own L2 cache
- Most caches are write-back, so it doesn't update main memory until the block gets evicted
- Problem for multithreaded code: If multiple threads are running with shared memory, how do we guarantee that the correct version of data is being used?
    - Can't access main memory anymore, if another cache has dirty data

Caching with multithreading

- Each core has its own L1 cache, and sometimes its own L2 cache
- Most caches are write-back, so it doesn't update main memory until the block gets evicted
- Problem for multithreaded code: If multiple threads are running with shared memory, how do we guarantee that the correct version of data is being used?
    - Can't access main memory anymore, if another cache has dirty data
- Three main cases
    - Two threads read
    - Two threads write
    - One thread reads, one thread writes

Caching with multithreading

- Synchronous reads:
  - Should be allowed; as long as no one changes data, things are safe
- Synchronous writes:
  - Once two different threads have conflicting data, it's difficult to return to a reasonable state
  - Example: git merge conflicts
  - Therefore, we should not allow these to happen
  - If threads attempt to cause this, we need to force a thread to evict
  - These evictions cause **coherence misses**; in a multithreaded system with shared memory, coherence misses can cause significant thrashing
- Multiple threads read, one writes
  - Could go the safe route and force evictions
    - But causes coherence misses
  - Ideally: Allow threads to read the written data from another core's cache, or set up a way for one cache to update other caches
    - Needs some kind of communication protocol

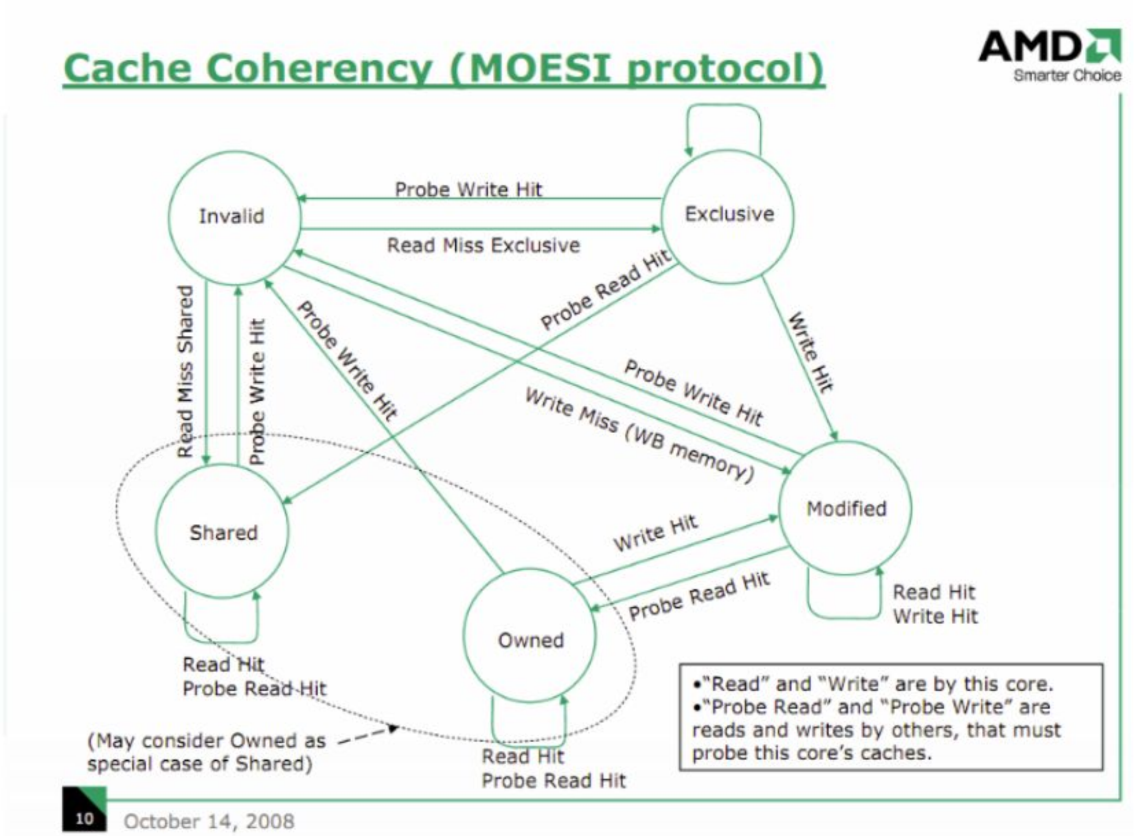
MOESI

- Solution: Set up a system where you can "snoop" on other caches to see if they have a cache block with the same tag
- Instead of just having a valid and dirty bit, have five different states for a block
  - Modified
  - Owned
  - Exclusive
  - Shared
  - Invalid
- Invalid: Same as valid bit off in regular cache; the block isn't in the cache
- Exclusive: Same as valid bit on, dirty bit off in regular cache
  - The block has been read, but not modified
  - Further, no other cache has this block
- Modified: Same as valid bit on, dirty bit on in regular cache
  - The block has been read, and modified
  - Further, no other cache has this block

MOESI

- Shared
  - The block is in some other cache
  - We haven't modified this block, and we're not allowed to make modifications
    - If we need to make modifications, evict the block from everyone's cache and move to modified
  - Allows for synchronous reads with at most one writer
- Owner
  - The block is in some other cache
  - We have modified the block, and are the only thread allowed to modify it
  - If we make modifications, it's our responsibility to tell all the other caches in shared state about these changes
  - Allows for writing while other threads read the same data

MOESI State Transitions



Cache Coherency (MOESI protocol)

- "Read" and "Write" are by this core.
- "Probe Read" and "Probe Write" are reads and writes by others, that must probe this core's caches.

10 October 14, 2008

Coherency Misses

- Since only one thread is allowed to have a dirty block at a time, two threads writing to the same block cause a lot of coherence misses
    - This is why the interleaved pragma omp was much slower than the blocked pragma omp
    - Another form of thrashing
- Therefore, when writing multithreaded code, we want to have each thread work on a separate block as much as possible
    - But also close enough so that the shared L3 cache gets hits

Conclusion

- Caching is a powerful tool that lets us save memory access time
- Tons of different ways to customize your cache, with different advantages/disadvantages
  - Direct-mapped to Fully Associative
  - Eviction policy
  - Write-back/Write-through
  - Multi-level caching
  - Shared vs independent caches
- Hardware design ends up trying to pick the best parameters so that the cache works well for most programs
- Software design can be improved by knowing your cache system and writing code that will be efficient.