CS 110
Computer Architecture
Lecture 18:
Caches Part III

Instructors:
Sören Schwertfeger & Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/21s/

School of Information Science and Technology SIST

ShanghaiTech University

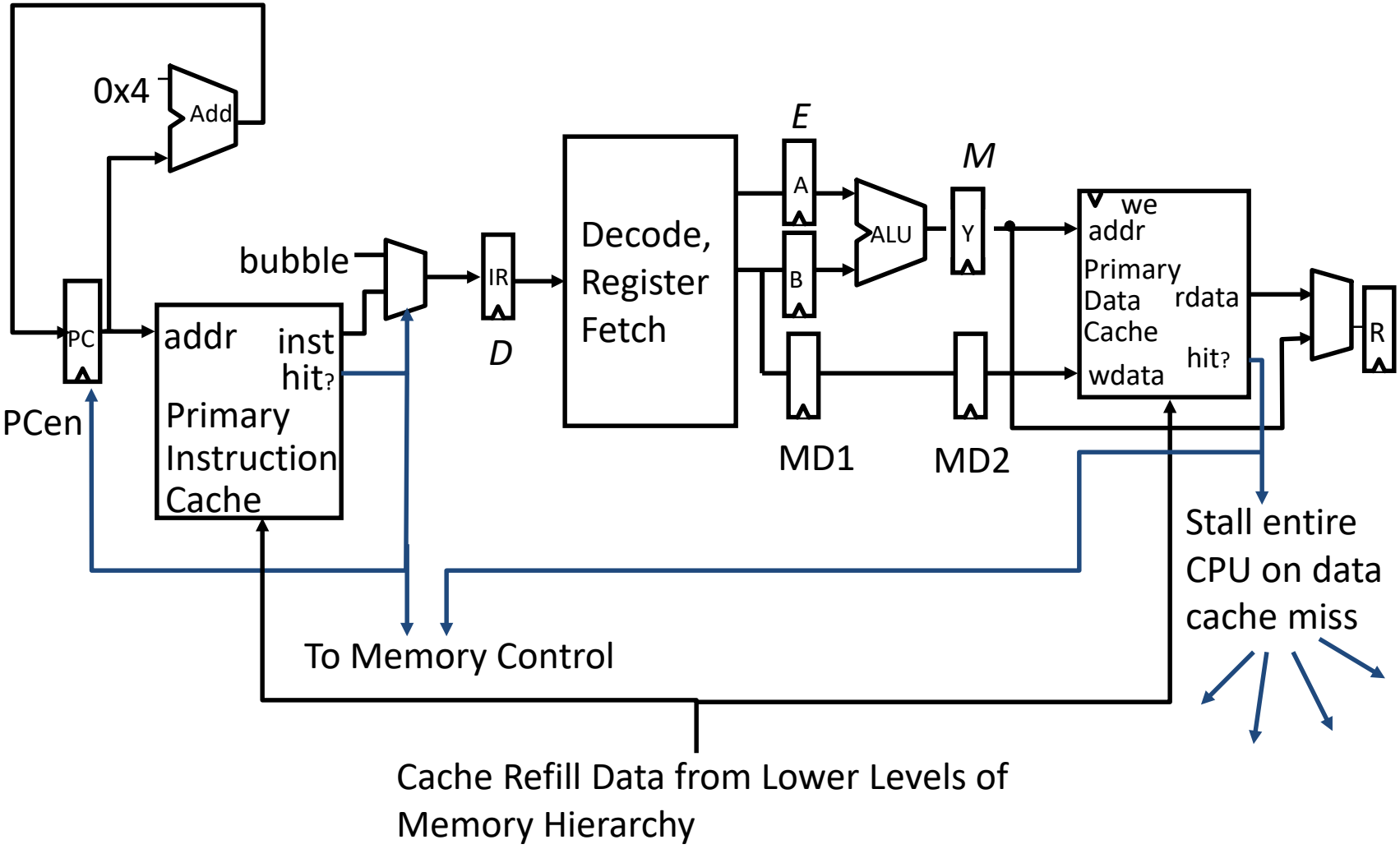
Slides based on UC Berkeley's CS61C

Cache Terms I

- ***Cache***:
  - A small and fast memory used to increase the performance of accessing a big and slow memory
  - Uses ***temporal locality:*** The tendency to reuse data in the same space over time
  - Uses ***spatial locality:*** The tendency to use data at nearby addresses
- Cache ***hit***: The address being fetched is in the cache ☺
- Cache ***miss:*** The address being fetched is not in the cache ☹
- ***Valid bit***: Is a particular entry valid
- Cache ***line flush***: Invalidate and flush one entry
  - e.g., clflush of x86, but newer clwb may not invalidate the entry
- Cache ***flush***: Invalidate all entries
  - e.g., wbinvd of x86

CPU-Cache Interaction

(5-stage pipeline)



Cache Terms II

- Cache ***level***:
  - The order in the memory hierarchy: L1$ is closest to the processor
  - L1 caches may only hold data (Data-cache, D$) or instructions (Instruction Cache, I$)
    - Most L2+ caches are "unified", can hold both instructions and data
- Cache ***capacity***:
  - The total # of bytes in the cache
- Cache ***line*** or cache ***block***:
  - A single entry in the cache
- Cache ***block size***:
  - The number of bytes in each cache line

Cache Terms III
Associativity

- ***Number of cache lines***:
  - Cache capacity / block size:
- Cache ***associativity***:
  - The number of possible cache lines a given address may exist in.
  - Also the number of comparison operations needed to check for an element in the cache
  - ***Direct mapped***: A data element can only be in one possible location (N=1)
  - ***N-way set associative***: A data element can be in one of N possible positions
  - ***Fully associative***: A data element can be at any location in the cache.
    - Associativity == # of lines
- Total # of cache lines == capacity of cache/line size

- Total # of lines in a set == # ways == N == associativity
- Total # of sets == # of cache lines / associativity

Victim Cache

- Conflict misses are a pain, but...
  - Perhaps a little associativity can help without having to be a fully associative cache
- In addition to the main cache...
  - Optionally have a very small (16-64 entry) ***fully associative*** "victim" cache
- Whenever we evict a cache entry
  - Don't just get rid of it, put it in the victim cache
- Now on cache misses...
  - Check the victim cache first, if it is in the victim cache you can just reload it from there

Cache Terms IV
Parts of the Address

- Address is divided into |***TAG***|***INDEX***|***OFFSET***|
- ***Offset***:
  - The lowest bits of the memory address which say where data exists within the cache line.
  - It is log2(line/block size)
  - So for a cache with 64B blocks it is 6 bits
- ***Index***:
  - The portion of the address which says where in the cache an address may be stored
  - Takes log2(# of cache lines / associativity) bits
  - So for a 4-way associative cache with 512 lines it is 7 bits
- ***Tag***: The portion of the address which must be stored in the cache to check if a location matches
  - # of bits of address - (# of bits for index + # of bits for offset)
  - So with 64-bit addresses it is 51-bit...

Cache Terms V
Writing

- ***Eviction***:
  - The process of removing an entry from the cache
- ***Write Back***:
  - A cache which only writes data up the hierarchy when a cache line is evicted
  - Instead set a ***dirty bit*** on cache entries
  - All i7 caches are ***write back***
- ***Write Through***:
  - A cache which always writes to memory
- ***Write Allocate***:
  - If writing to memory ***not in the cache*** fetch it first
  - i7 L2 is Write Allocate
- ***No Write Allocate***:
  - Just write to memory without a fetch
  - i7 L1 is no write allocate

Replacement Policy

In an associative cache, which line from a set should be evicted when the set becomes full?

- Random
- Least-Recently Used (LRU)
  - LRU cache state must be updated on every access
  - True implementation only feasible for small sets (2-way)
  - Pseudo-LRU binary tree often used for 4-8 way
- First-In, First-Out (FIFO) a.k.a. Round-Robin
  - Used in highly associative caches
- Not-Most-Recently Used (NMRU)
  - FIFO with exception for most-recently used line or lines

This is a second-order effect. Why?

Replacement only happens on misses

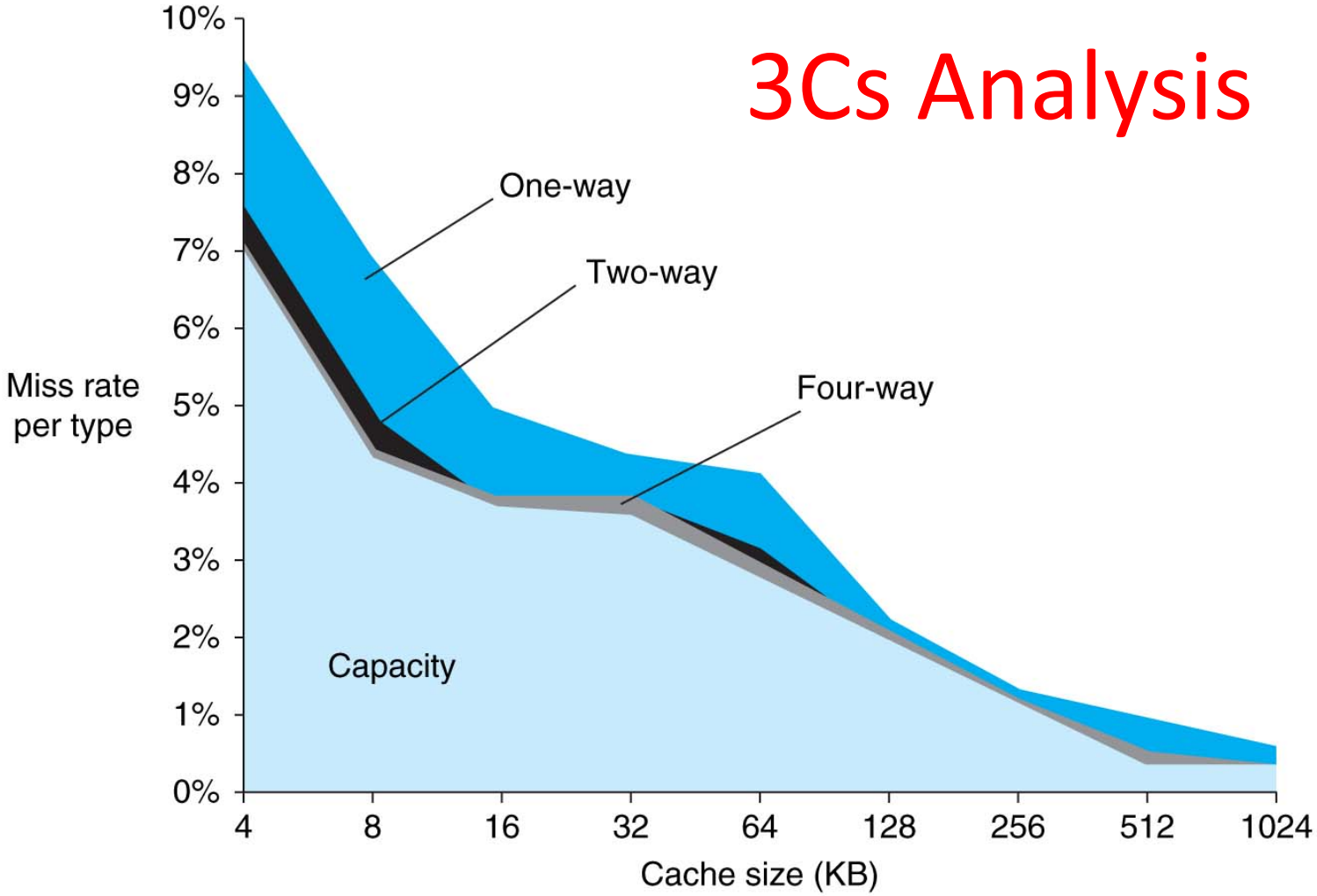
Cache Terms VI
Cache Performance

- ***Hit Time***:
  - Amount of time to return data in a given cache: depends on the cache
  - i7 L1 hit time: 4 clock cycles
- ***Miss Penalty***:
  - Amount of ***additional*** time to return an element if its not in the cache: depends on the cache
- ***Miss Rate***:
  - Fraction of a ***particular program's*** memory requests which miss in the cache
- Average Memory Access Time (***AMAT***):
  - Hit time + Miss Rate * Miss Penalty

Understanding Cache Misses: The 3Cs

- Compulsory (cold start or process migration, 1st reference):
  - First access to block impossible to avoid; small effect for long running programs
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)
- Capacity:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size (may increase access time)
- *Conflict (collision):*
  - *Multiple memory locations mapped to the same cache location*
  - *Solution 1: increase cache size*
  - *Solution 2: increase associativity (may increase access time)*

3Cs Analysis



- Three sources of misses (SPEC2000 integer and floating-point benchmarks)
  - Compulsory misses 0.006%; not visible
  - Capacity misses, function of cache size
  - Conflict portion depends on associativity and cache size

Improving Cache Performance

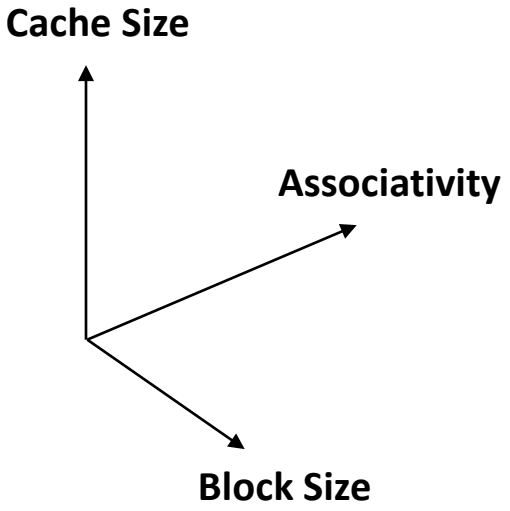
AMAT = Time for a hit + Miss rate x Miss penalty

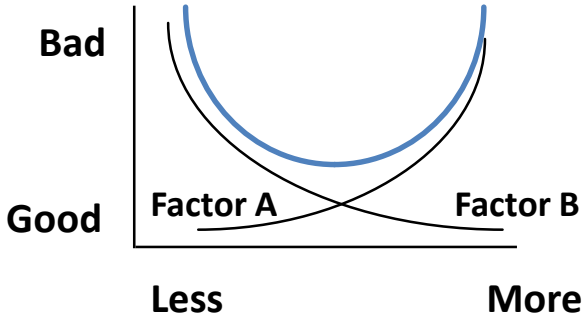
- Reduce the time to hit in the cache
  - E.g., Smaller cache
- Reduce the miss rate
  - E.g., Bigger cache
- Reduce the miss penalty
  - E.g., Use multiple cache levels

Cache Design Space

Computer architects expend considerable effort optimizing organization of cache hierarchy – big impact on performance and power!

- Several interacting dimensions
  - Cache size
  - Block size
  - Associativity
  - Replacement policy
  - Write-through vs. write-back
  - Write allocation
- Optimal choice is a compromise
  - Depends on access characteristics
    - Workload
    - Use (I-cache, D-cache)
  - Depends on technology / cost
- Simplicity often wins





Increasing Associativity?

- Hit time as associativity increases?
  - Increases, with large step from direct-mapped to >=2 ways, as now need to mux correct way to processor
  - Smaller increases in hit time for further increases in associativity
- Miss rate as associativity increases?
  - Goes down due to reduced conflict misses, but most gain is from 1->2->4-way with limited benefit from higher associativities
- Miss penalty as associativity increases?
  - Unchanged, replacement policy runs in parallel with fetching missing line from memory

Increasing #Entries?

- Hit time as #entries increases?
  - Increases, since reading tags and data from larger memory structures
- Miss rate as #entries increases?
  - Goes down due to reduced capacity and conflict misses
  - *Architects rule of thumb: miss rate drops ~2x for every ~4x increase in capacity (only a gross approximation)*
- Miss penalty as #entries increases?
  - Unchanged

At some point, increase in hit time for a larger cache may overcome the improvement in hit rate, yielding a decrease in performance

Increasing Block Size?

- Hit time as block size increases?
  - Hit time unchanged, but might be slight hit-time reduction as number of tags is reduced, so faster to access memory holding tags
- Miss rate as block size increases?
  - Goes down at first due to spatial locality, then increases due to increased conflict misses due to fewer blocks in cache
- Miss penalty as block size increases?
  - Rises with longer block size, but with fixed constant initial latency that is amortized over whole block

Another Cache: Branch Predictor

- In our simple pipeline, we assume branches-not-taken
  - Always start fetching the next instruction
- If a branch or jump is taken...
  - Then we have to kill the non-taken instructions so they don't cause side effects
- But both branches and jumps are PC relative...
  - So if we can quickly look at the instruction and decide *'eh, probably taken/not'*, we can compute the new location for the PC if we can guess right
  - Which for **jal** we always can, but branches we need to guess
- Idea: ***branches*** have temporal locality!
  - Loops: **for (x = 0; x < n...)**
  - Rare conditionals: **if (err) ...**

A Simple Branch Predictor

- Have an N entry, direct-mapped memory
  - E.g., a 1024x1b memory
- If fetched instruction is a branch...
  - Check if the bit for pc[12:2] is set in IF...
  - If so, set next PC to PC + branch offset fetched (in ID probably, if not IF)
  - Set bit in pipeline to say “branch predicted-taken”
- When actually evaluating branch in EX...
  - Set pc[12:2] in the branch predictor to branch taken/not-taken status
  - If branch taken but predicted not-taken
    - ***Kill untaken instructions***
  - If branch not taken but predicted taken
    - ***Kill predicted instructions***

Where to do this?

- If we could, do it in IF
  - Now on correct predictions we will always be right
- If we can't, do it in ID
  - First non-taken instruction will be fetched regardless, so we need more complex control logic in determining which to kill, but !.
- This does complicate the pipeline a fair bit, but worth it!
  - Let's assume the branch comparison is done in EX…
  - If we can predict in IF in the 5 stage pipeline:
    - Correct predicted branches -> ***no stalls***
    - Incorrect prediction -> 2 stalls for killed instructions
  - If we can predict in ID:
    - Correct predicted taken branch -> 1 stall
    - Correct predicted not-taken branch -> 0 stalls
    - Incorrect predicted taken branch -> 2 stalls
    - Incorrect predicted not-taken branch -> 2 stalls

Improving it slightly...

- How about 2 bits:
  - Each entry starts at 01...
  - If taken, increment with saturating arithmetic (so max is 11)
  - If not taken, decrement by one (so min is 00)
- If the upper bit is 1, assume the branch is taken
  - Now a function with a commonly taken loop will only mispredict once rather than twice
- Miss penalty for a mispredicted branch: The # of instructions that got terminated because of the wrong prediction
  - E.g., on a dual-issue, 10-deep 2x superscalar like a Raspberry Pi or smartphone: Probably ~10 instructions
  - On a modern x86? It could be 20+
- Can then try even fancier predictors...
  - But we get into ***diminishing returns:***
    - The simple-smart thing (e.g. a two bit branch predictor) is a big win...
    - But trying to get fancier no longer helps nearly as much

Approximate Cache...

- The data caches are exact:
  - They will return an answer that is exactly what is asked for
- But this branch predictor is approximate:
  - It can make mistakes due to aliasing:
    Its not actually storing the full address as a tag to check
- Sometimes its OK to be wrong
  - So data structures that are this way are also particularly interesting...
  - E.g. Bloom filter https://en.wikipedia.org/wiki/Bloom_filter

A related cache: return target location...

- Observation:
  - On RISC-V, you call a function with **jal** or **jalr** (object oriented) with the return set to **ra**
  - And you return with a **jalr** with the source register as **ra**
- So let's maintain a small stack in hardware...
  - Whenever we see **jal** or **jalr** writing to **ra**: We write PC+4 into the stack
  - Whenever we see **jalr** reading from **ra**: We predict the top of this stack as the next PC, and pop this stack
- Result: We should ***always*** correctly predict a function return address...
  - Works as long as we don't exceed the stack depth: once we hit that we will start getting misses

And A Final Related Cache: Branch Target Buffer

- Function calls using **jal** we will never mispredict on RISC-V
  - Since they are all PC relative we can do the add in the decode where we change our PC prediction
- But so much today is object-oriented programming which uses **jalr**: C++ and Java object calls are equivalent to calling pointers to functions
  - **foo.bar()** is implemented as something like this:

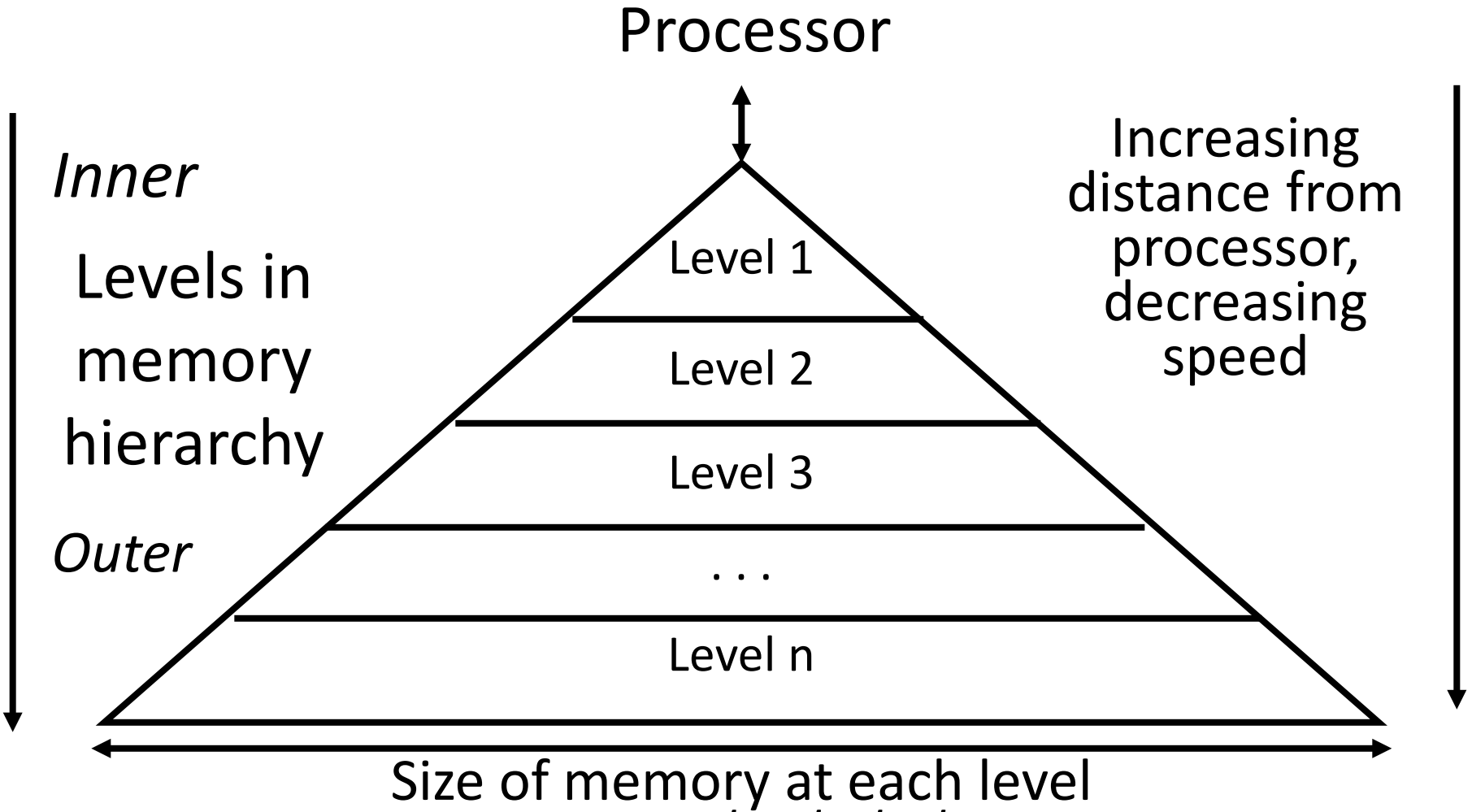
```
lw t0 0(a0)  # Get the pointer to the "virtual function table" in the object
lw t0 8(t0)  # Get the pointer to the function to actually call
jalr ra t0   # Do a JALR to call bar(),
             # with the object foo as the first implicit argument
```

- So cache this as well:
  - On a **jalr** which writes to **ra** rather than **x0**.
    Look in a small cache for the address to predict to based on current PC
  - When evaluating the jump, set the value in this cache to the address used
- It is the x86 equivalent of this cache that is part of one of the Spectre vulnerabilities

How to Reduce Miss Penalty?

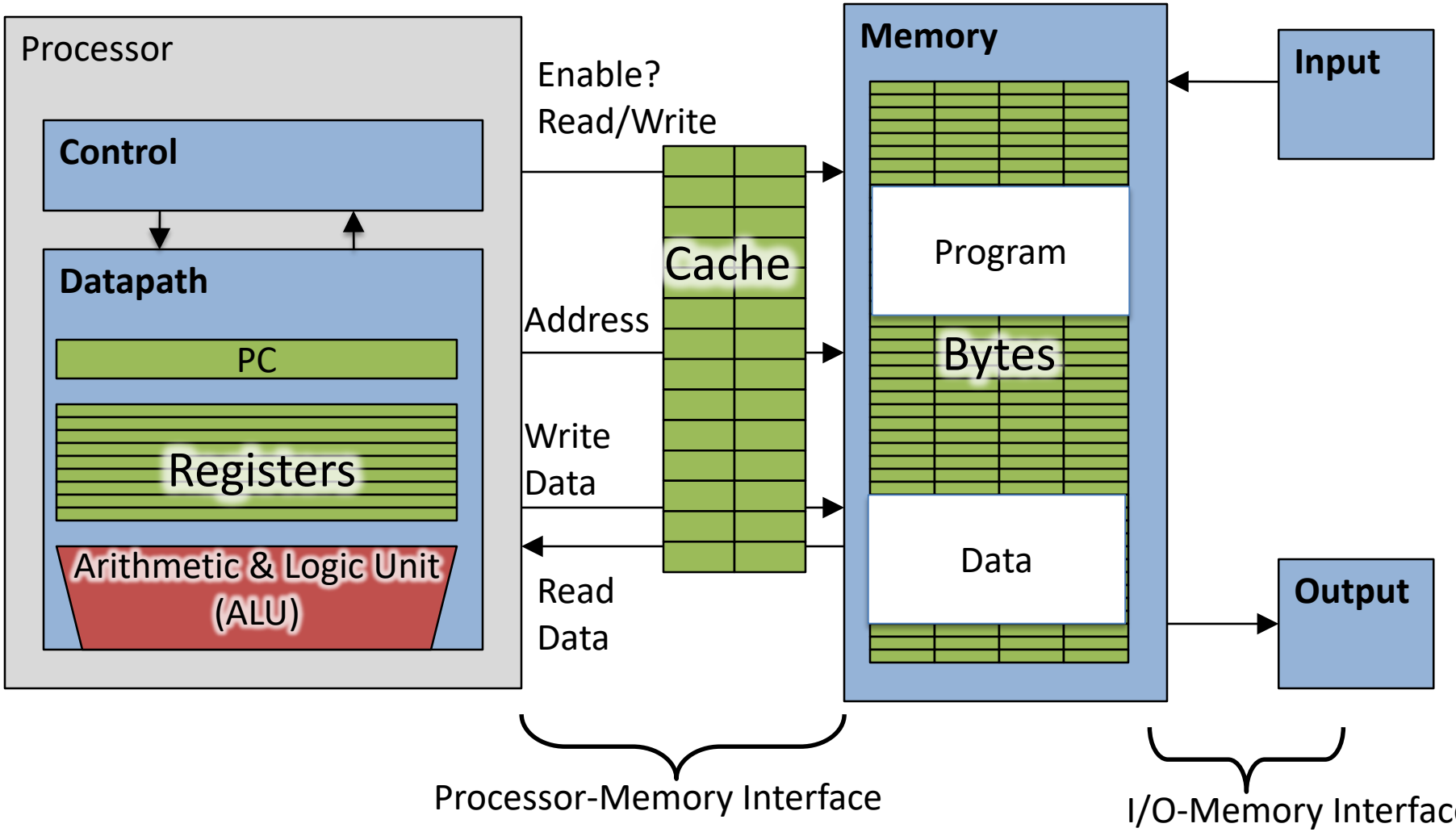
- Could there be locality on misses from a cache?
- Use multiple cache levels!
- With Moore's Law, more room on die for bigger L1 caches and for second-level (L2) cache
- And in some cases even an L3 cache!
- IBM mainframes have ~1GB L4 cache off-chip.

Review: Memory Hierarchy

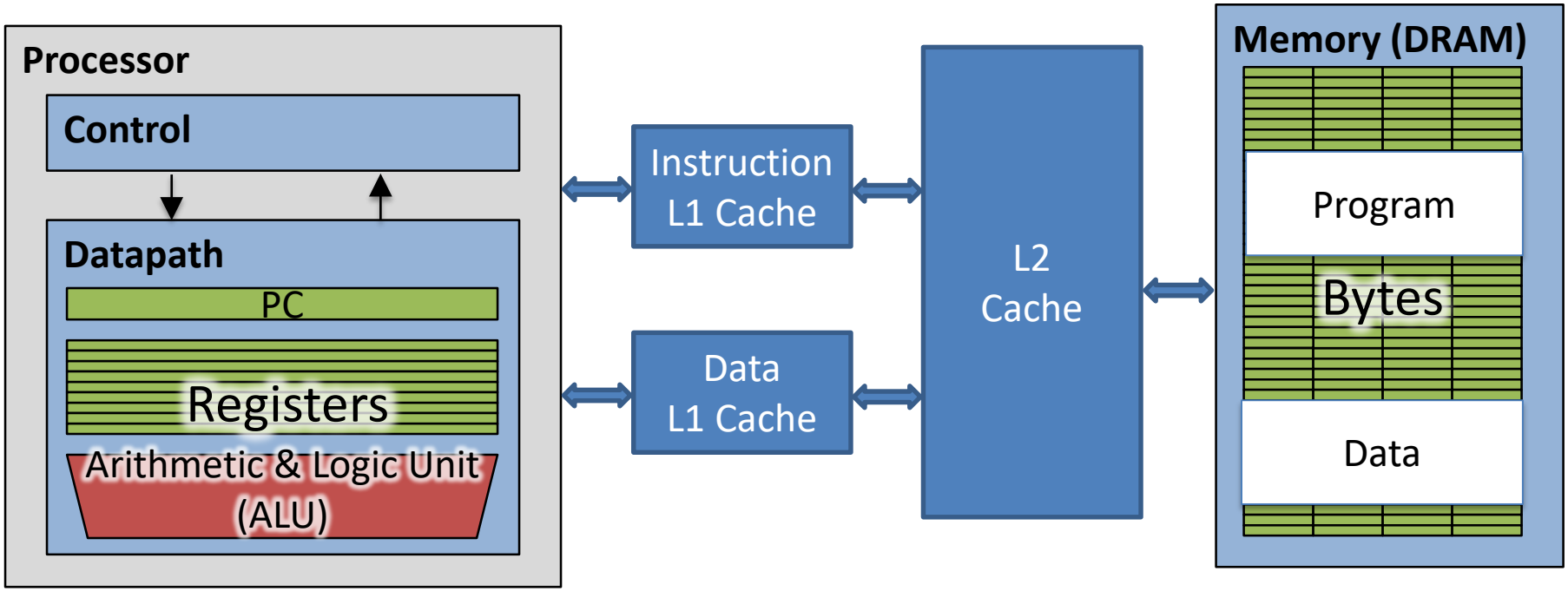


As we move to outer levels the latency goes up and price per bit goes down.

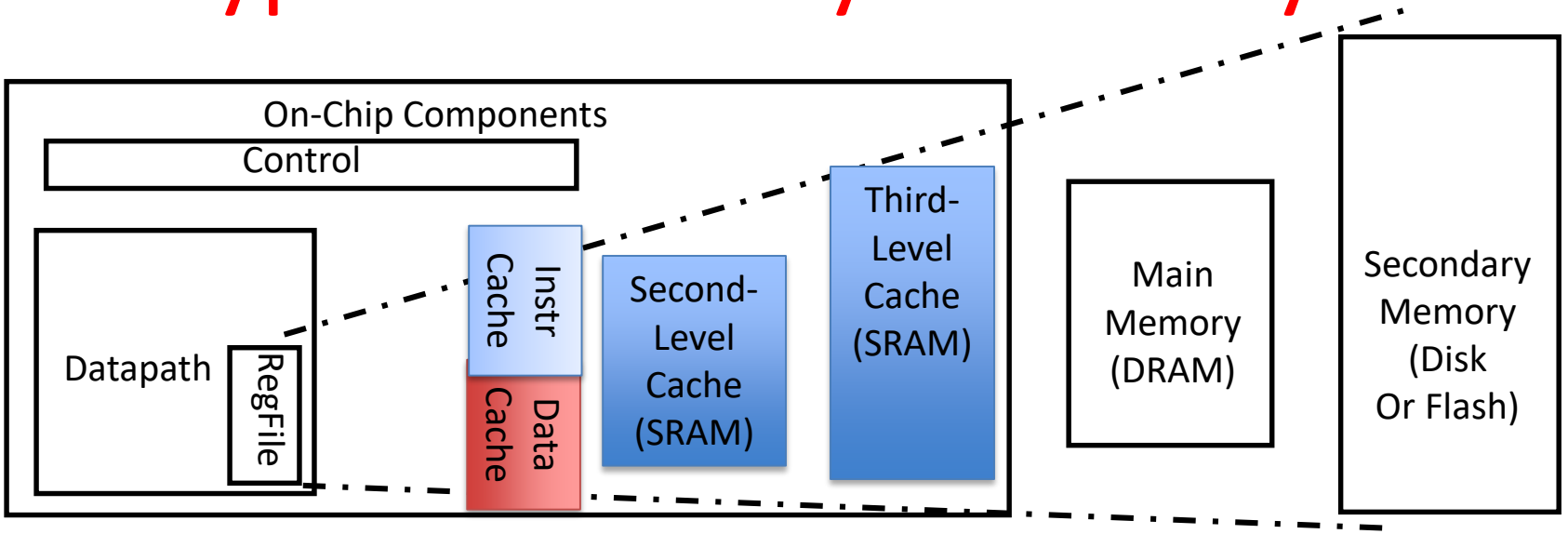
Adding Cache to Computer



L1 and L2 Caches



Typical Memory Hierarchy



Speed (cycles):	½'s	1's	10's	100's	1,000,000's
Size (bytes):	100's	10K's	M's	G's	T's
Cost/bit:	highest	←		→	lowest

- Principle of locality + memory hierarchy presents programmer with ≈ as much memory as is available in the *cheapest* technology at the ≈ speed offered by the *fastest* technology

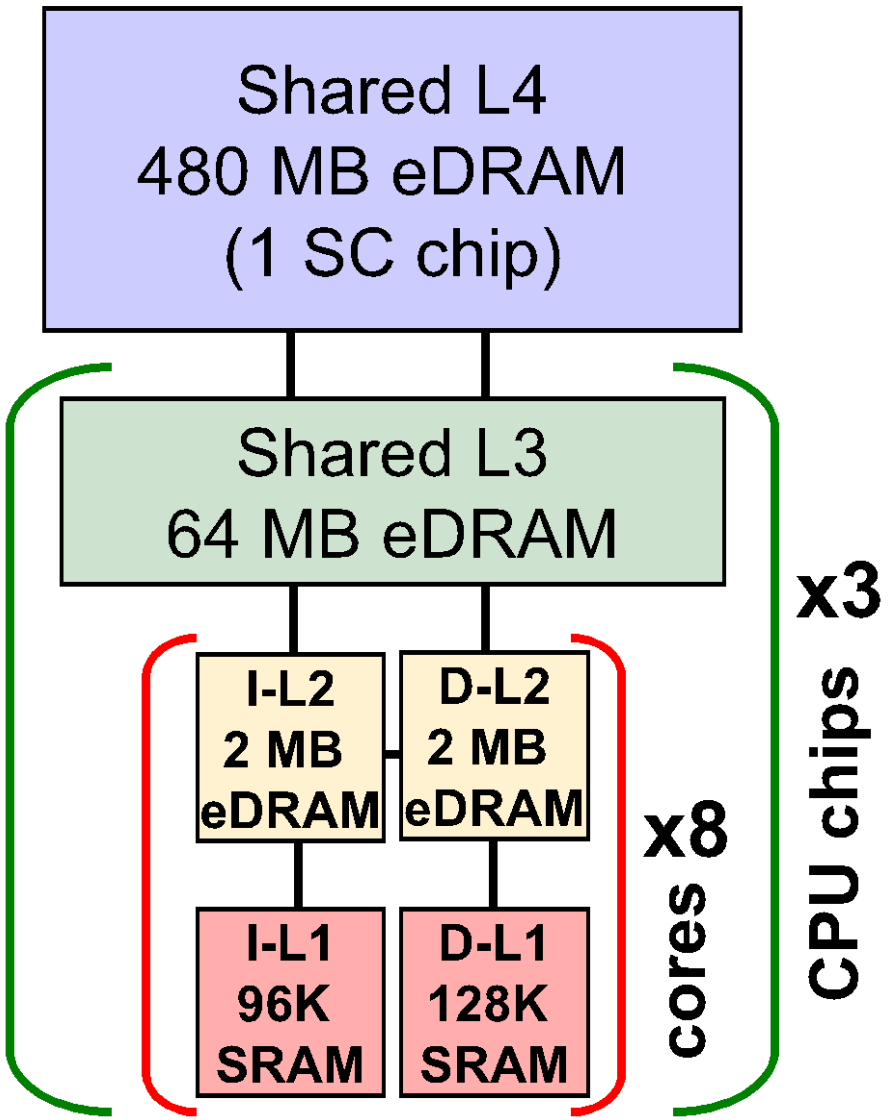
How is the Hierarchy Managed?

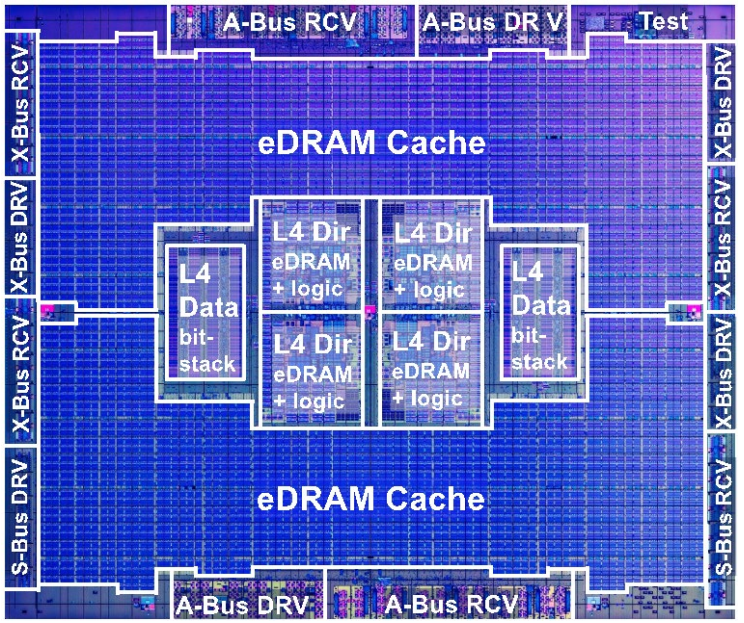
- registers ↔ memory
  - By compiler (or assembly level programmer)
- cache ↔ main memory
  - By the cache controller hardware
- main memory ↔ disks (secondary storage)
  - By the operating system (virtual memory)
  - Virtual to physical address mapping assisted by the hardware ('translation lookaside buffer' or TLB)
  - By the programmer (files)

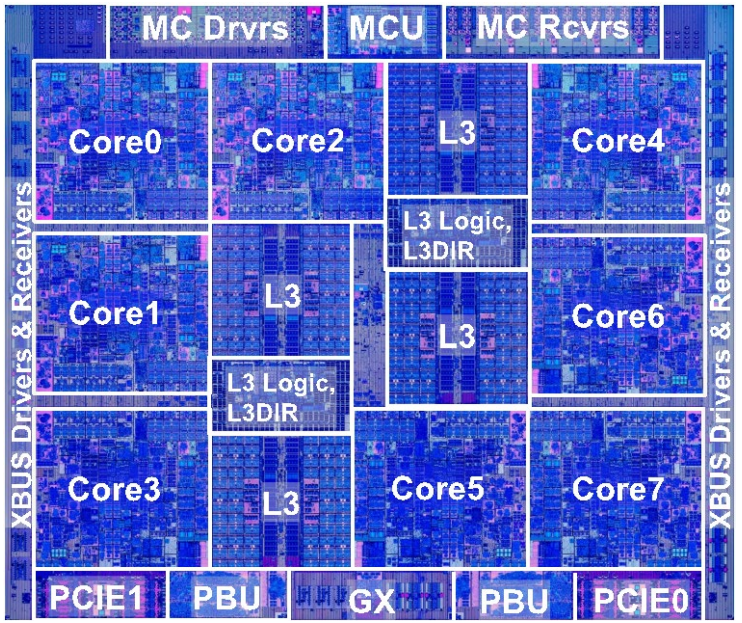
2015 IBM CPU

- z13 designed in 22nm SOI technology with seventeen metal layers, 4 billion transistors/chip
- 8 cores/chip, with 2MB L2 cache, 64MB L3 cache, and 480MB L4 off-chip cache.
- 5GHz clock rate, 6 instructions per cycle, 2 threads/core
- Up to 24 processor chips in shared memory node

IBM z13 Memory Hierarchy

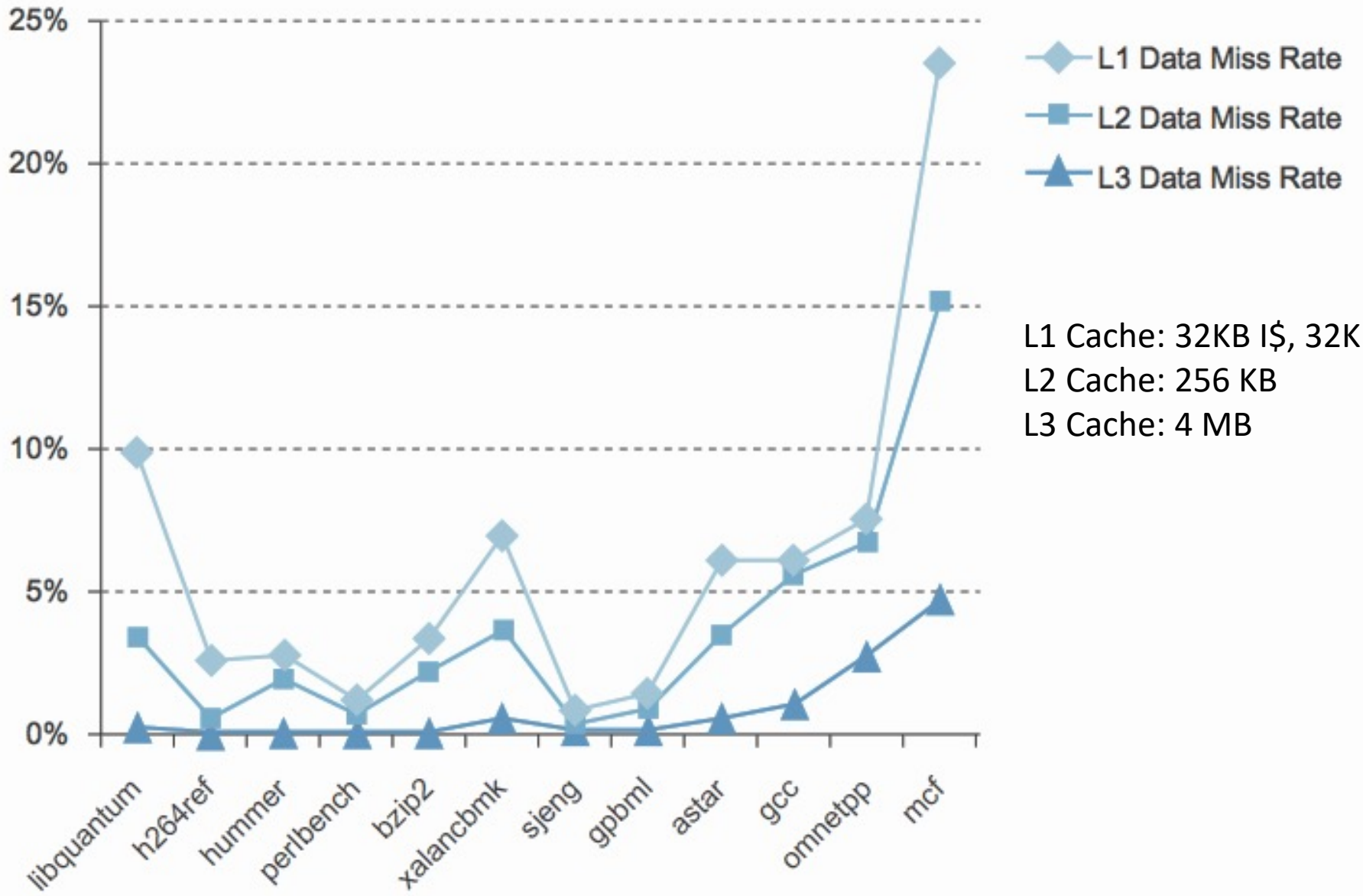






Local vs. Global Miss Rates

- *Local miss rate* – the fraction of references to one level of a cache that miss
- Local Miss rate L2$ = L2$ Misses / L1$ Misses
  = L2$ Misses / total_L2_accesses
- *Global miss rate* – the fraction of references that miss in all levels of a multilevel cache
  - L2$ local miss rate >> than the global miss rate



L1 Cache: 32KB I$, 32KB D$
L2 Cache: 256 KB
L3 Cache: 4 MB

FIGURE 5.47 The L1, L2, and L3 data cache miss rates for the Intel Core i7 920 running the full integer SPECCPU2006 benchmarks.

Local vs. Global Miss Rates

- *Local miss rate* – the fraction of references to one level of a cache that miss
- Local Miss rate L2$ = $L2 Misses / L1$ Misses
- *Global miss rate* – the fraction of references that miss in all levels of a multilevel cache
  - L2$ local miss rate >> than the global miss rate
- Global Miss rate = L2$ Misses / Total Accesses
  = (L2$ Misses / L1$ Misses) × (L1$ Misses / Total Accesses)
  = Local Miss rate L2$ × Local Miss rate L1$

- AMAT = Time for a hit + Miss rate × Miss penalty
- AMAT = Time for a L1$ hit + (local) Miss rate L1$ × (Time for a L2$ hit + (local) Miss rate L2$ × L2$ Miss penalty)

Characteristic	Intel Nehalem	AMD Opteron X4 (Barcelona)
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KB each for instructions/data per core	64 KB each for instructions/data per core
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L1 hit time (load-use)	Not Available	3 clock cycles
L2 cache organization	Unified (instruction and data) per core	Unified (instruction and data) per core
L2 cache size	256 KB (0.25 MB)	512 KB (0.5 MB)
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	Not Available	9 clock cycles
L3 cache organization	Unified (instruction and data)	Unified (instruction and data)
L3 cache size	8192 KB (8 MB), shared	2048 KB (2 MB), shared
L3 block size	64 bytes	64 bytes
L3 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L3 hit time	Not Available	38 (?)clock cycles

CPI/Miss Rates/DRAM Access SpecInt2006

Name	CPI	Data Only L1 D cache misses/1000 instr	Data Only L2 D cache misses/1000 instr	Instructions and Data DRAM accesses/1000 instr
perl	0.75	3.5	1.1	1.3
bzip2	0.85	11.0	5.8	2.5
gcc	1.72	24.3	13.4	14.8
mcf	10.00	106.8	88.0	88.5
go	1.09	4.5	1.4	1.7
hmmer	0.80	4.4	2.5	0.6
sjeng	0.96	1.9	0.6	0.8
libquantum	1.61	33.0	33.1	47.7
h264avc	0.80	8.8	1.6	0.2
omnetpp	2.94	30.9	27.7	29.8
astar	1.79	16.3	9.2	8.2
xalancbmk	2.70	38.0	15.8	11.4
Median	1.35	13.6	7.5	5.4

In Conclusion, Cache Design Space

- Several interacting dimensions
  - Cache size
  - Block size
  - Associativity
  - Replacement policy
  - Write-through vs. write-back
  - Write-allocation
- Optimal choice is a compromise
  - Depends on access characteristics
    - Workload
    - Use (I-cache, D-cache)
  - Depends on technology / cost
- Simplicity often wins