Bonus Lecture: Caches are cool

EECS 370 – Introduction to Computer Organization – Winter 2023

EECS Department
University of Michigan in Ann Arbor, USA

Bonus lecture

- This lecture contains non-testable material.
  - It's just bonus stuff that I think is interesting.
    - Hopefully you do too.

Topics

- Memory system overview
- Different types of caches
  - Hash, victim, and skew caches
- The math of locality
  - Locality and stack distance
  - Visualizing locality and associativity
- Multiprocessor systems and caches
  - A bit “MESI”

Why is cache important (a review)?

- CPUs are fast, DRAM isn't.
  - Accessing a random date element from DRAM takes about 100ns
    - That's not just DRAM latency, but everything thrown together
  - CPUs can have periods around 0.25ns (4 GHz)
  - Modern CPUs can expect to execute multiple instructions per clock if things are otherwise going well.
    - So in the time of a single memory access, we could have executed 100s of instructions (maybe even 1000!)
- So we want to go to memory as little as possible.

Memory system overview

- Multi-level caches
  - Modern processors have a significant memory hierarchy
  - Something that's pretty typical (Sunny Cove numbers) is:
    - L1 I-cache, 32KB 8-way
    - L1 D-cache, 48KB, 12-way
    - L2 cache, 512-1280KB, 8-20 way
    - L3 cache, 2048 KB per core, 16-way cache among multiple cores
- Let's look at a die photo

10nm ESF/Intel 7 Alder Lake die shot (~209mm²) from Intel via Andreas Schilling on Twitter:
https://twitter.com/aschilling/status/1453391035577495553

Die shot interpretation by Locuza, October 2021

Things are even more complex

- We have store queues
  - Where we keep stores queued up to go to memory if there are loads to work on
- Stride predictors
  - Basically an attempt, based on memory access patterns seen so far, to guess what data the processor may ask for and start getting it before the processor asks for it.
- And other issues/techniques
  - TLBs, victim buffers/caches, etc.

How do we hide latencies?

- This is more of a CPU question.
  - "How do we cope with the fact that memory can be slow?"
- The easiest technique is to just add more pipeline stages.
  - If the I-cache is going to take 2 cycles, just have fetch take 2 cycles.
    - You should appreciate that this will increase the CPI due to hazards.
  - This really only hides L1 *hits*!
- A more tricky technique is to have the processor have something else to do while waiting for the data.
  - Have more than one thread running at a time (470/482)
  - Find something not "in the shadow of the load" that misses.
    - Out-of-order processors (470)
- Not today's topic, but worth thinking about!

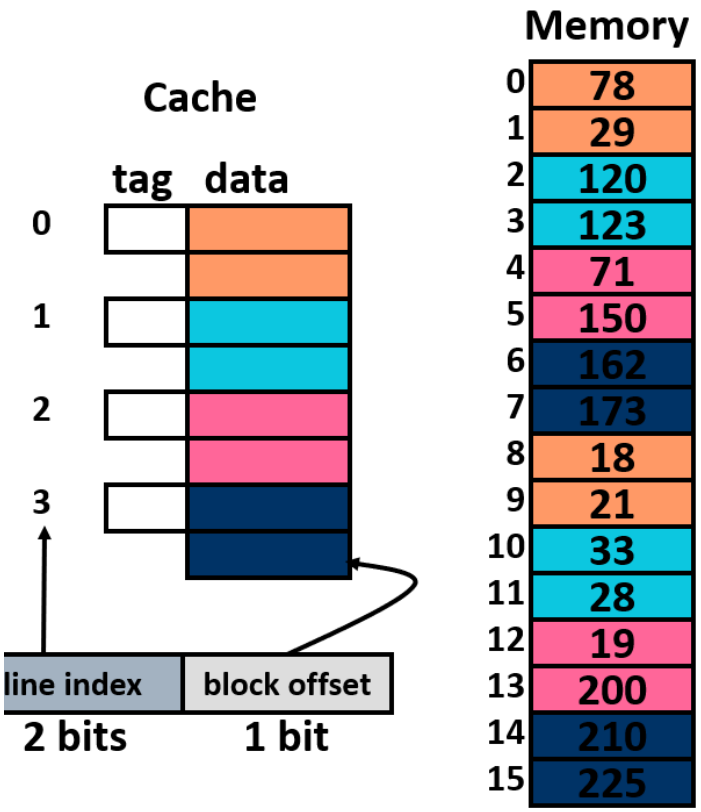
Different types of caches

Making direct-mapped caches not be quite as bad

- As we discussed in class, direct-mapped caches tend to get a pretty poor hit rate.
  - There may be “excessive conflict” or a “hot set”.
    - That is two or more memory locations that map to the same cache set may be heavily used.
    - And so they kick each other out all the time.
    - That creates a big miss rate.
  - What causes the excessive conflict?
    - Sometimes it’s just bad luck
    - Sometimes it’s the nature of the code.
- So we could either try to reduce the excessive conflict or find a way to reduce the impact.

Reduce the conflict of a DM$

- Spatial locality says that things near each other in memory (close addresses) are likely to be accessed near each other in time.
  - So things which are far apart in memory are not likely to be accessed near each other in time.
  - We take advantage of this by having things which conflict in a cache be spaced out from each other.
- But real programs often see conflicts beyond random!
  - We'd expect less than random.



Reduce the conflict of a DM$: Hash cache

- Idea:
  - Grab some bits from the tag and use them, as well as the old index bits, to select a set.
  - Simplest version would be if N sets, grab the 2N lowest order bits after the offset and XOR them in groups of 2.

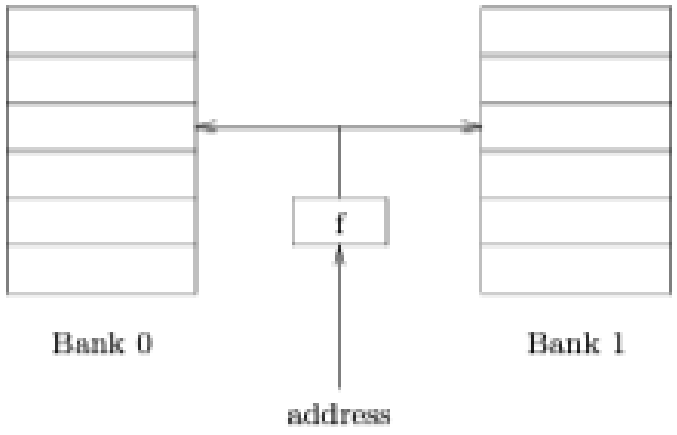
- In practice this often reduces the "hot set" problem.
  - It does keep two blocks that map to a given set fairly far apart in memory much of the time.
    - See hash cache, one's complement cache.

Reduce the impact of a DM$: Victim cache

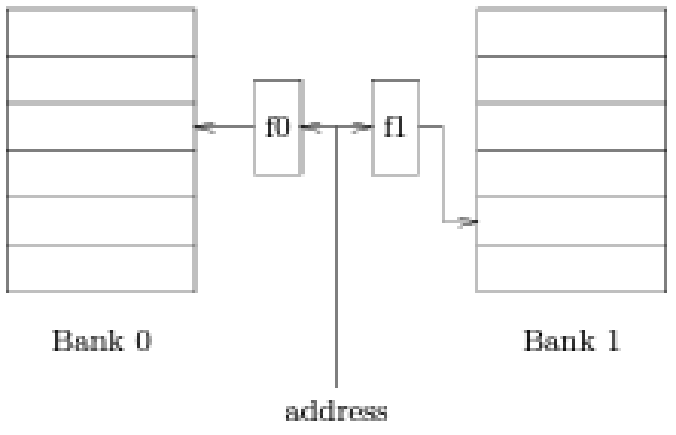
- If we have a "hot set" problem we are kicking out data from one set, only to need it soon thereafter.
- Idea:
  - Let's add a small, fully associative cache to the direct-mapped cache.
  - We'll search it in parallel with the DM cache.
    - It can be faster than the DM cache because it will be really small (maybe 4 lines while the DM$ might have 128+ lines)
    - So it shouldn't slow down the DM cache much at all since we can do it in parallel.
  - We call this small cache a "victim cache" because a line that gets evicted is a "victim".
    - These greatly improve a DM cache's performance.

A "wacky idea"

- Skew cache
  - Basically like the hash cache, but each way uses a different index!



2-way associative cache



Skew cache

- This gets the advantages of a hash cache, but also, weirdly, acts like a >2 way associative cache. (more on that later)

The math of locality

Locality review

- Temporal locality
  - We tend to access memory locations we've accessed recently.
  - We take advantage of temporal locality by keeping recently accessed data in our cache.
    - **LRU** replacement.

- Spatial locality
  - We tend to access memory locations *near* locations we've accessed recently.
  - We mostly take advantage of spatial locality by keeping the data in the cache in **blocks** bigger than a word.

Let's measure locality

- Consider two different memory streams
  - A: 0x0000, 0x0020, 0xf000, 0x0000, 0x0002
  - B: 0x0000, 0x0020, 0xf000, 0x0100, 0x5502
  - Which has a greater degree of locality?
  - Can we somehow quantify that?

Some basics

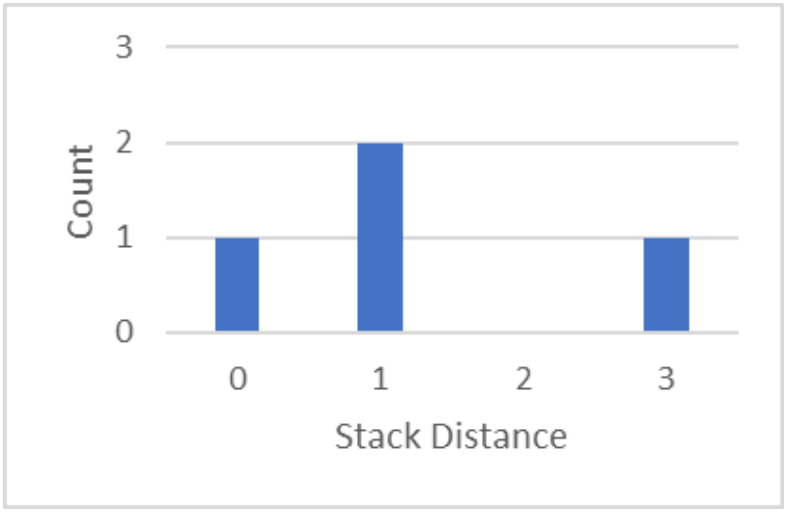
- Let's do this for a fixed block size and just look at block numbers.
  - So if we have 16-byte blocks, we just drop the bottom 4 bits of the address to get the block number.
    - So address 0x5502 becomes block number 0x550.

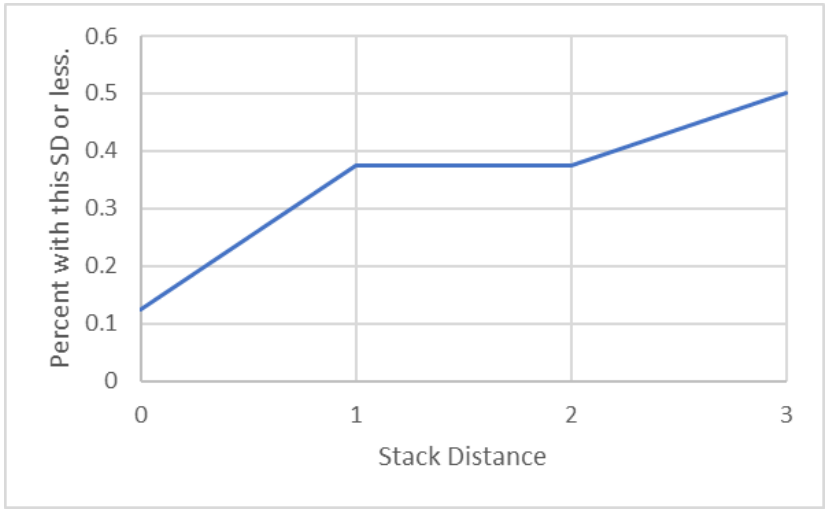
- Now let's just check how many *unique* accesses we've had since the last time we saw the block before.
  - We'll define this so that a block we've never seen before was seen an "infinite" time ago.

Block reference stream	0	1	0	0	1	2	4	0
Unique intervening accesses	∞	∞	1	0	1	∞	∞	3

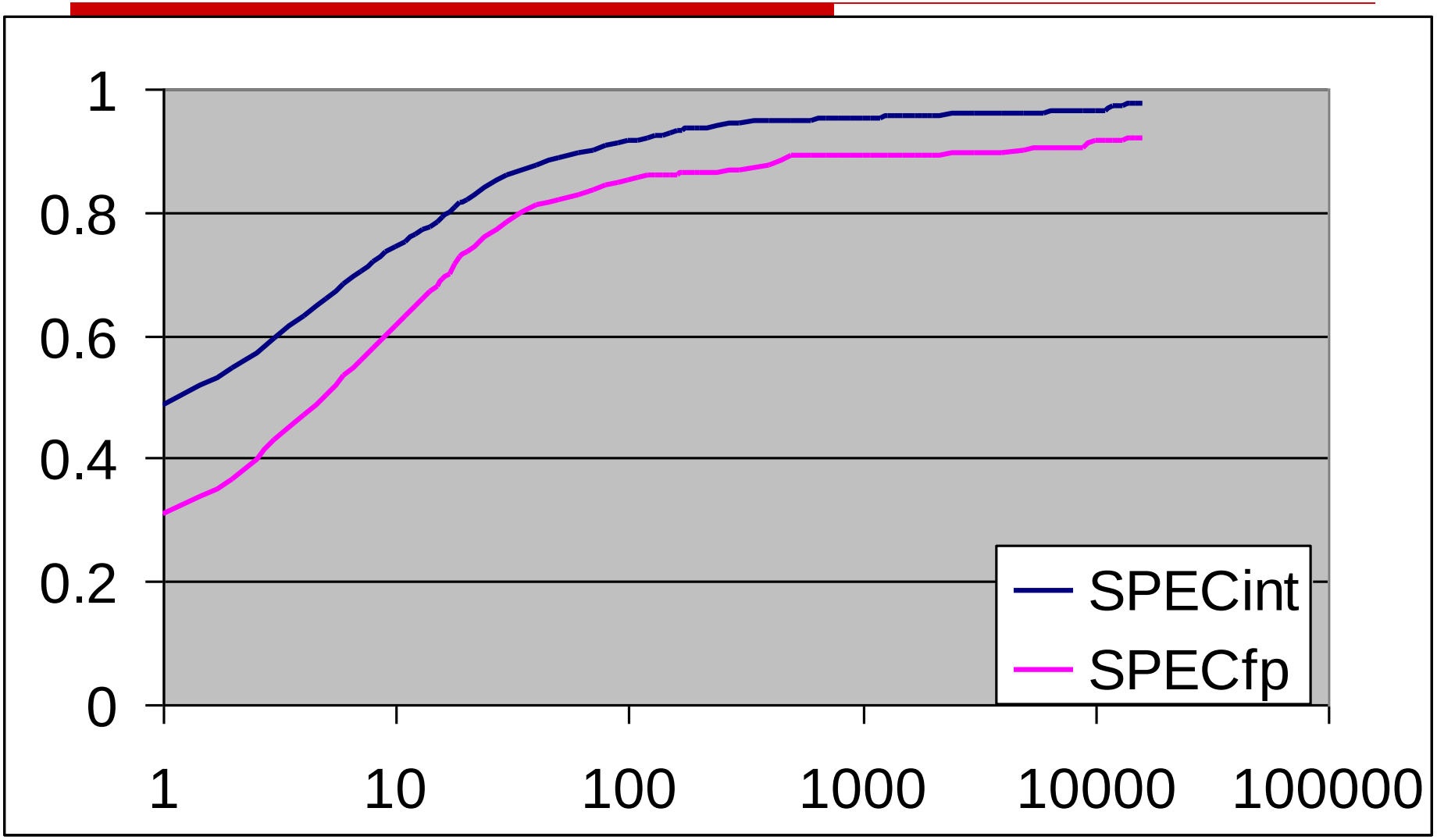
Graph the distribution

Block reference stream	0	1	0	0	1	2	4	0
Stack Distance	∞	∞	1	0	1	∞	∞	3

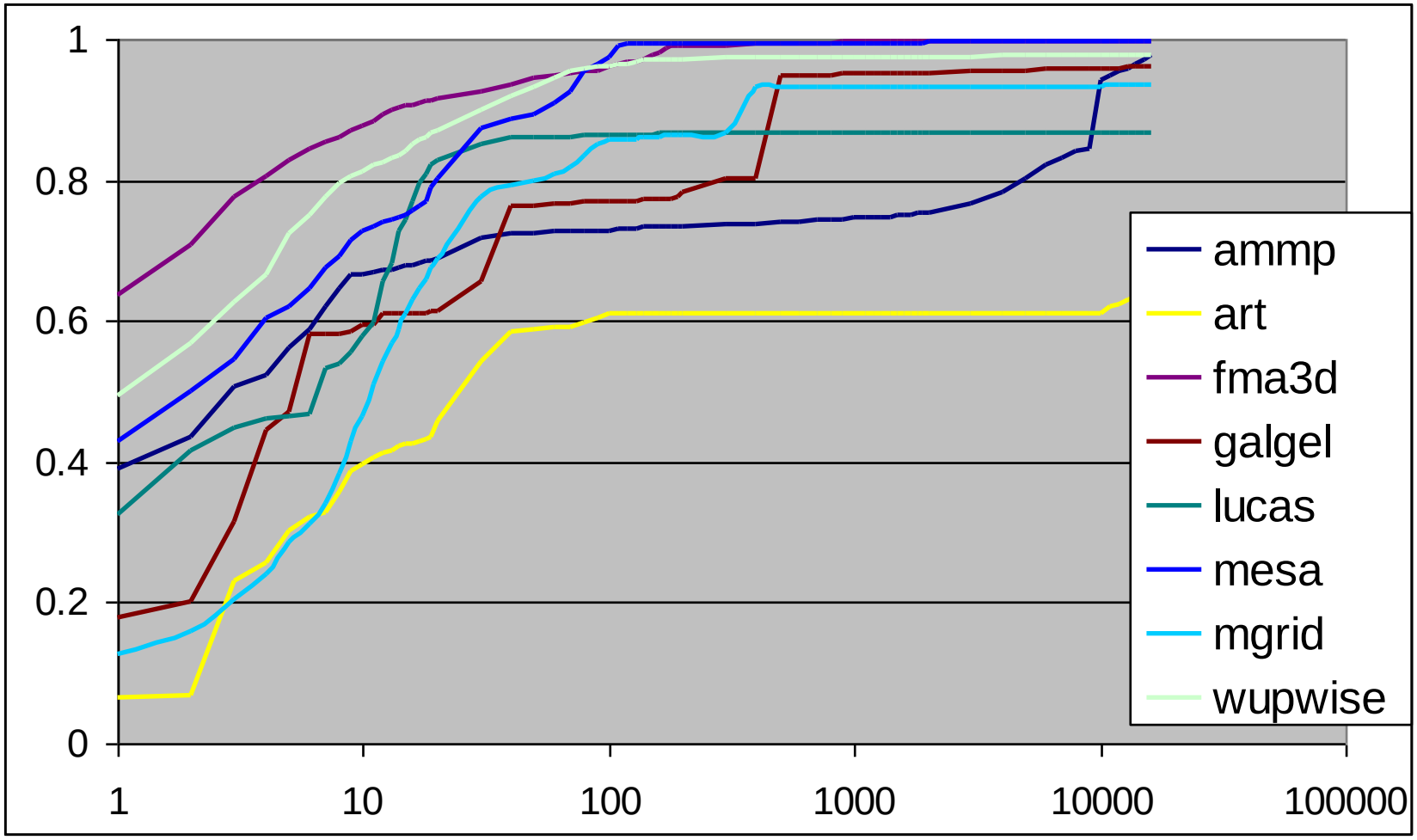




Stack distances of sets of real programs



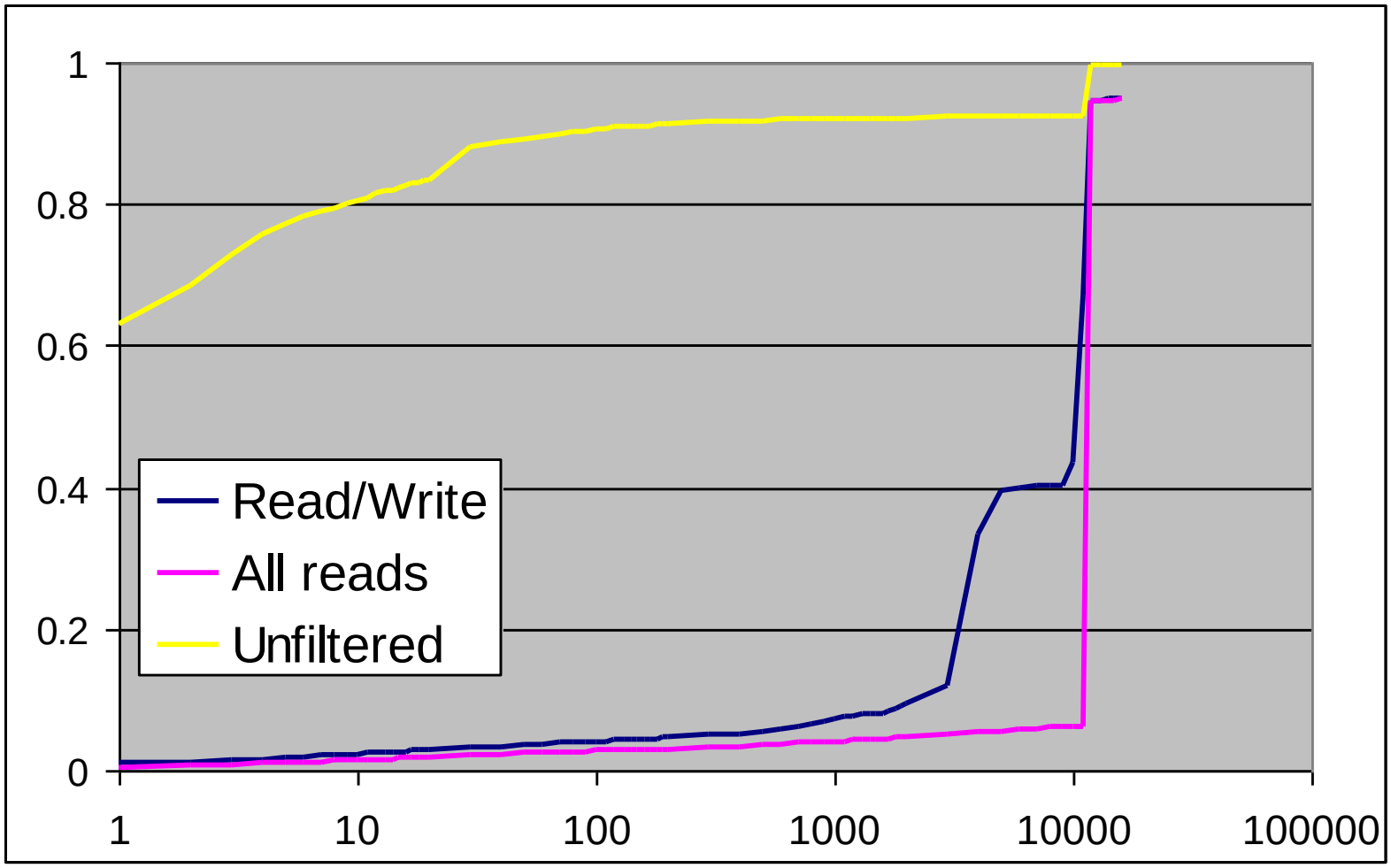
Stack distances of individual programs from SPECfp



Why is this interesting/useful?

- We can see that some things have more or less locality than others.
- Can visualize working set size and generally understand the locality of program at a glance

Example:
Observing the filtering of locality after an L1 cache.



Visualizing how caches interact with locality

- A fully-associative cache with 4 blocks will:
  - Get a hit on any memory access that has a stack distance of <4.
  - Get a miss on any memory access that has a stack distance of ≥ 4
  - Why?
- What about a direct-mapped cache with 4 lines?
  - What hit rate on a memory access with a stack distance of 0?
  - 1?
  - 2?
  - General formula for a DM$ hit for an access with a stack distance of n?

$$\left(\frac{3}{4}\right)^n$$

Direct-mapped caches

- For direct-mapped cache with 4 lines the formula for a DM$ hit for an access with a stack distance of n is

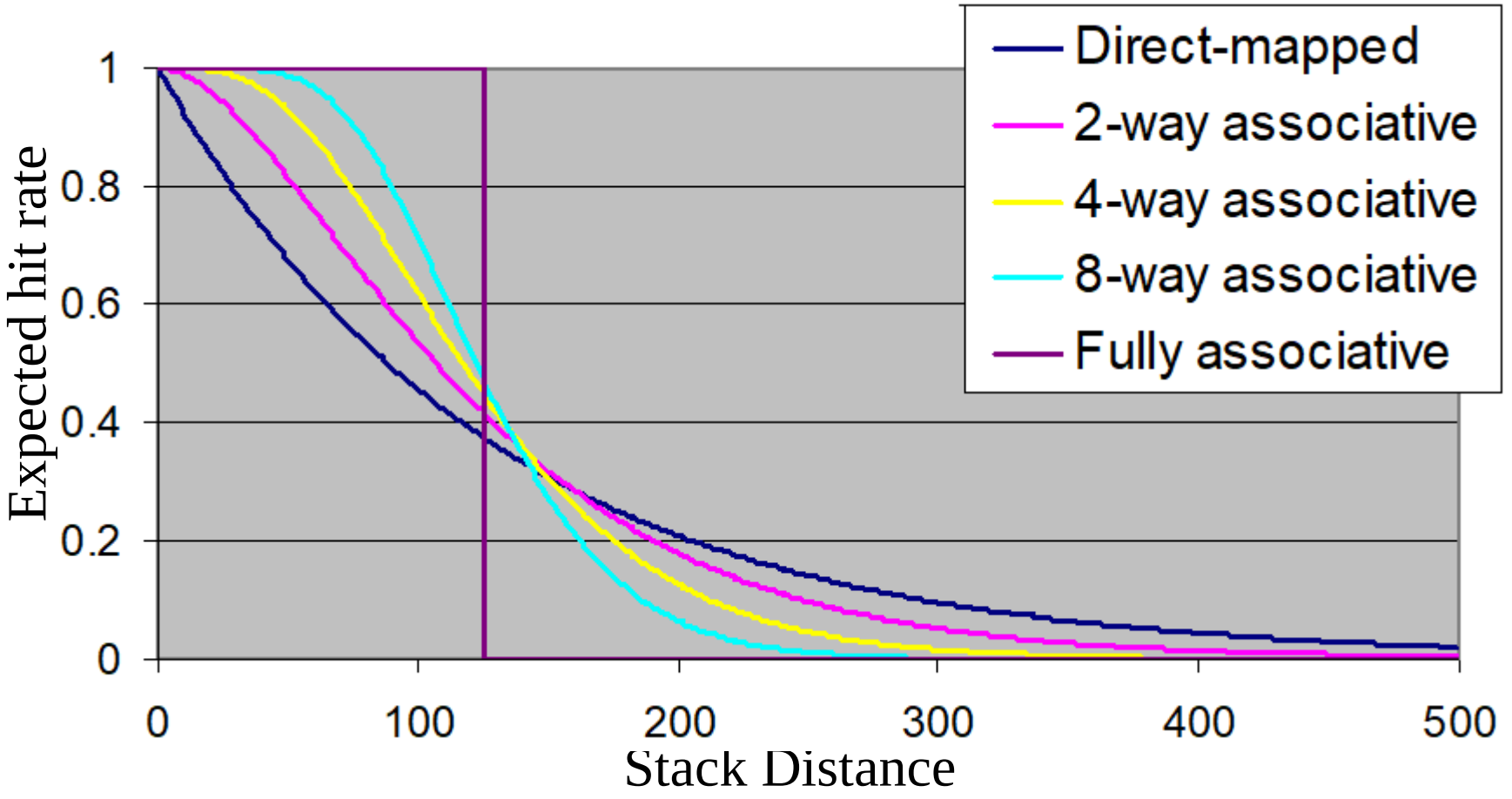
$$\left(\frac{3}{4}\right)^n$$

- What about a DM$ with m lines?

$$\left(\frac{m-1}{m}\right)^n$$

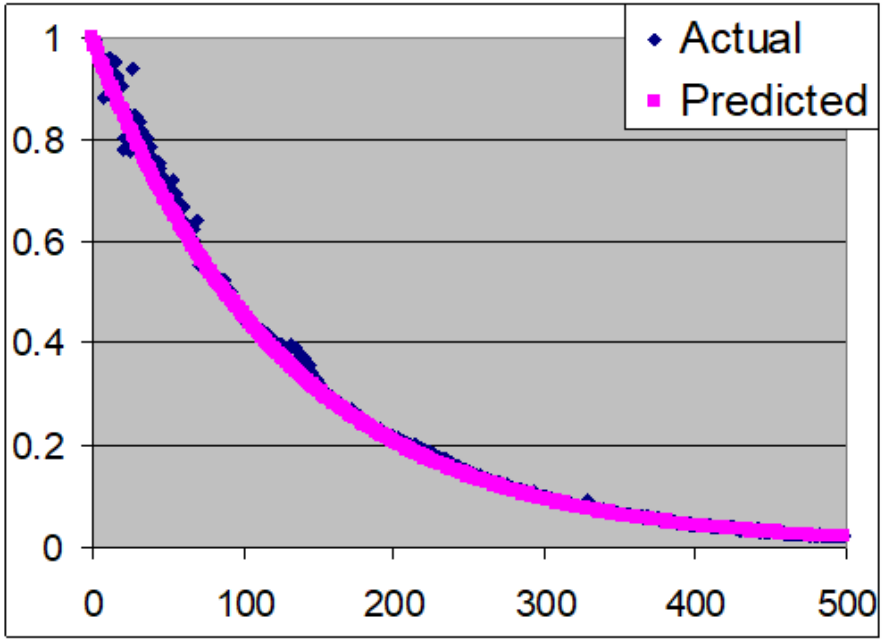
- One could imagine doing the same for a set-associative cache.

Expected hit rate at a given stack distance for a 128-line cache

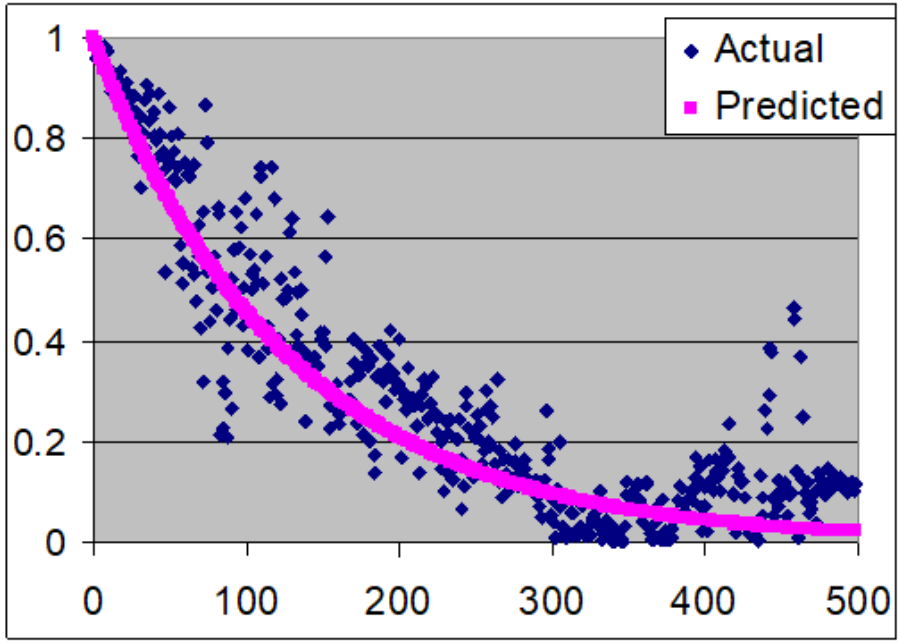


How close is this to reality?

SPECint



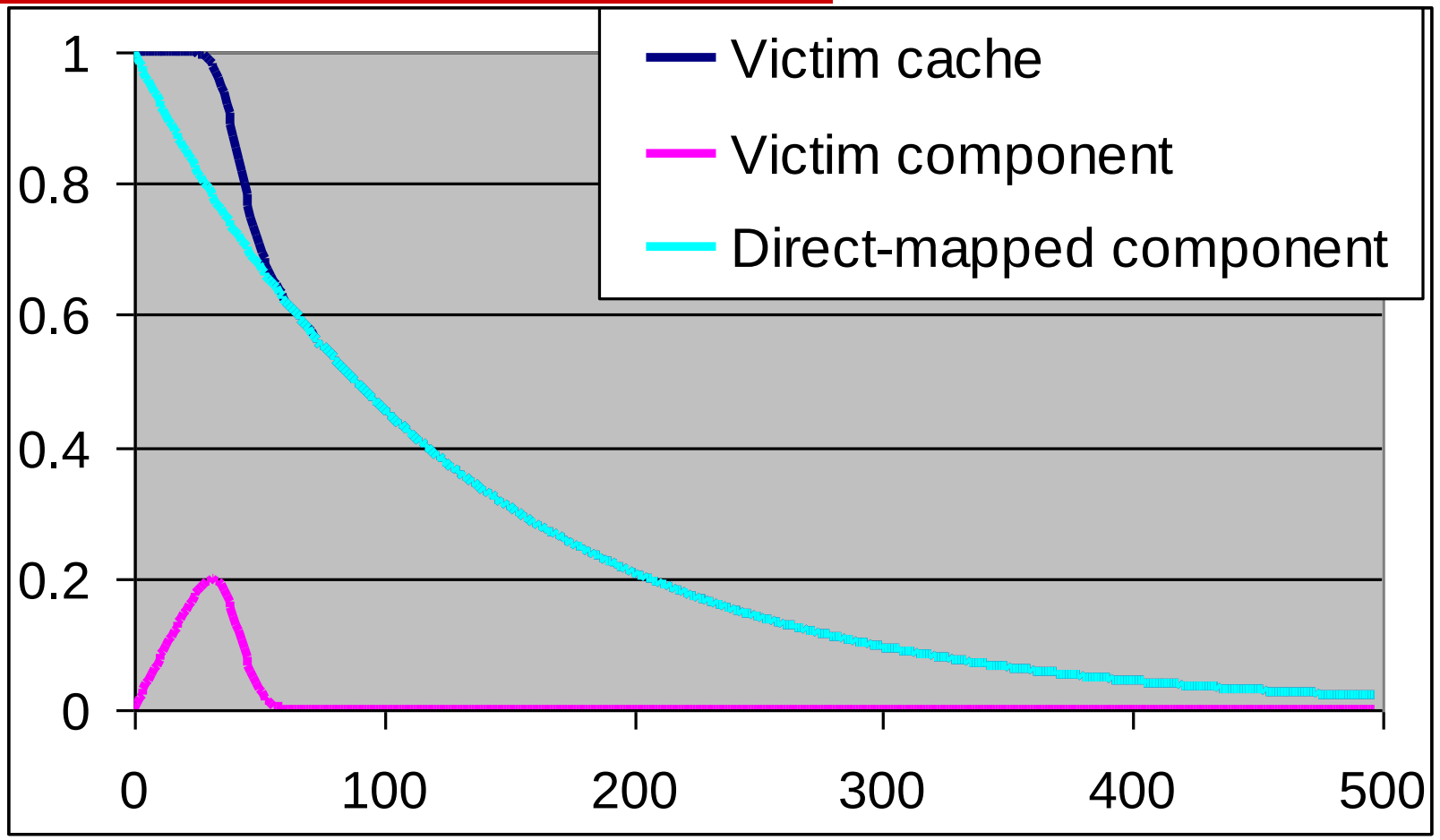
SPECfp



Why is this interesting?

- It is possible to see exactly *why* more associative caches do better than less associative caches
  - It also becomes possible to see when they do worse.
  - The area under the curve is always equal to the number of cache lines.
- It provides an *expectation* of performance.
  - If things are worse, there must be excessive conflict.
  - If things are better, it is likely due to spatial locality.
    - Conflict between blocks should be less than random!

Example: Visualizing how a victim cache helps



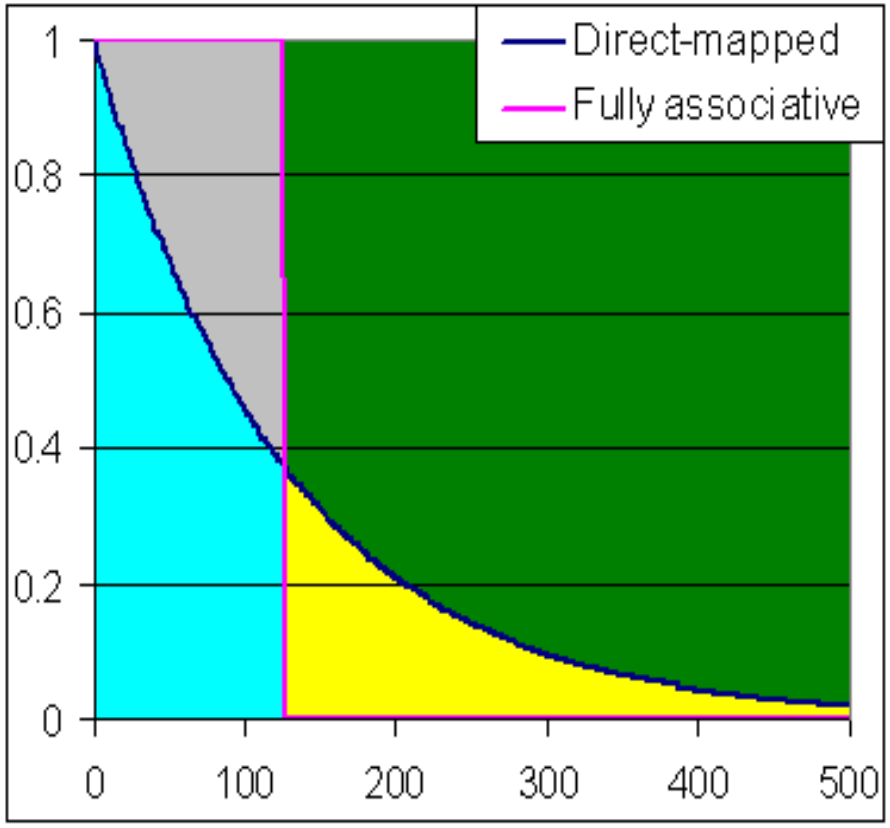
128-line direct-mapped component and a 6-line victim component

One last example: the 3Cs model.

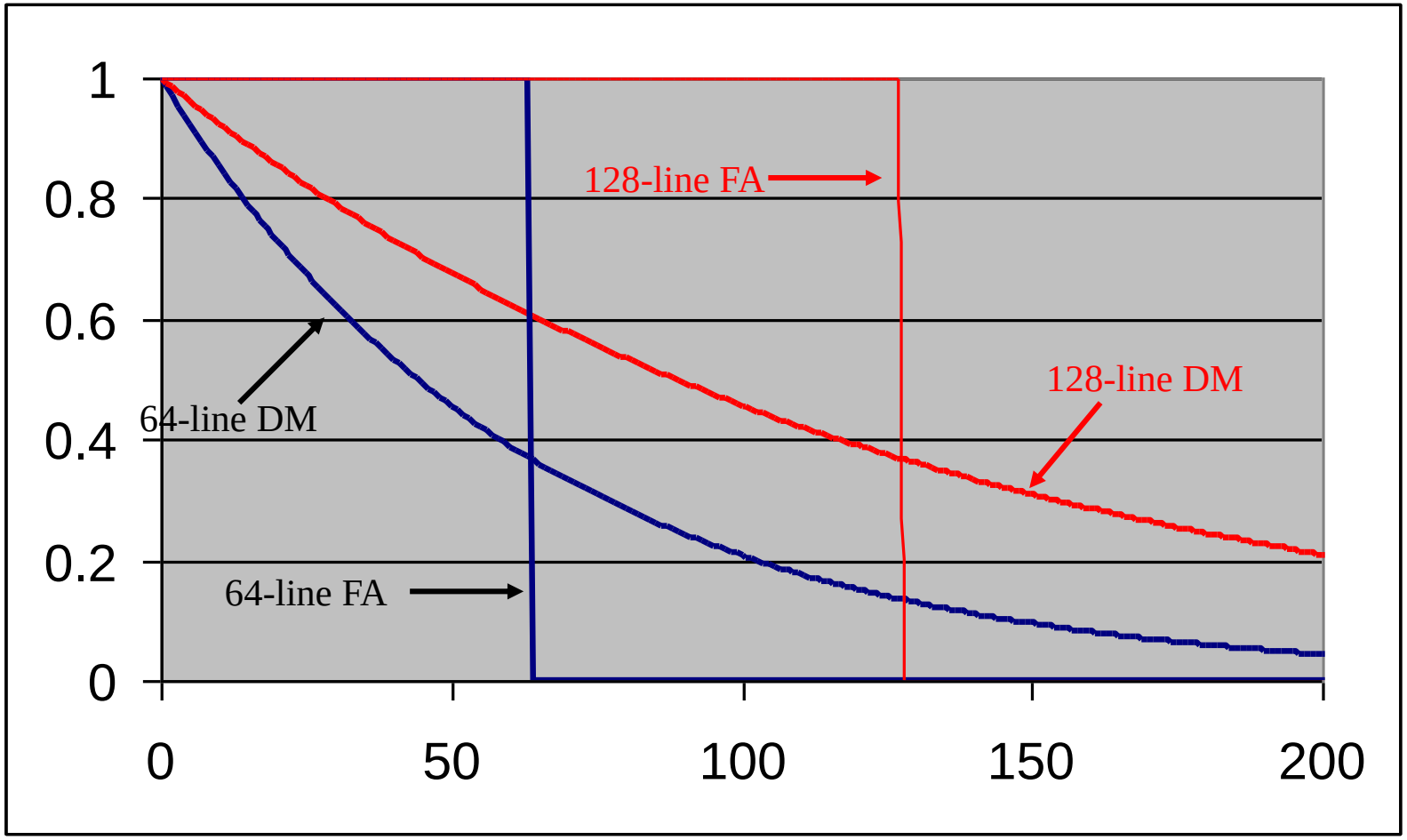
The 3C's model

The 3'Cs model describes misses as conflict, capacity or compulsory by comparing a direct-mapped cache to a fully-associative cache.

- Those accesses in the gray area are *conflict* misses.
- The in the green area are *capacity* misses.
- The blue area is where both caches get a hit.
- The yellow area is ignored by the 3C's model. Perhaps a "conflict hit"?



Thinking about 3Cs a bit more…



That's it.

- Hope you found it somewhat interesting.