EECS 470

Cache overview

4 Hierarchy questions

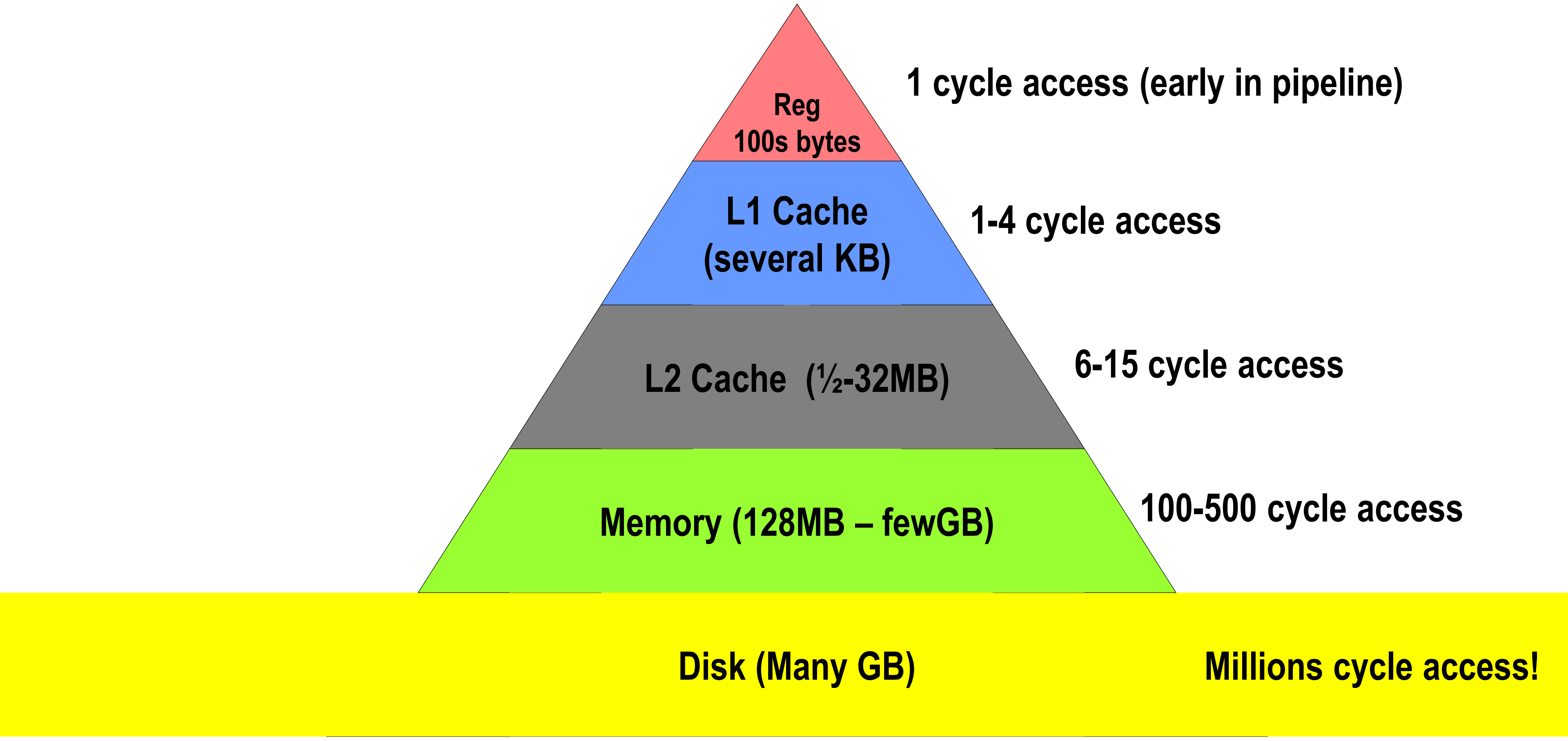
More on Locality

Lecture 11 & 12: Caches – Winter 2024

Class project

- Project restrictions
  - The I-cache and D-cache size is limited to the size it has in what we gave you.
  - Can’t have more CDBs then degree of super scalarness.
  - Must use our multiplier (P2)
- Milestone 1:
  - It is due on Tues 3/5
  - Should have high-level design done ASAP.
  - Module you hand in should be:
    - Self testing, well written.
  - Be aware of sample (single) RS on the website.

Cache Design 101



Cache[1]

- **1 a :** a hiding place especially for concealing and preserving provisions or implements
  b : a secure place of storage
- **3 :** a computer memory with very short access time used for storage of frequently used instructions or data -- called also *cache memory*

[1]From **Merriam-Webster** on-line

Locality of Reference

- Principle of Locality:
  - Programs tend to reuse data and instructions near those they have used recently.
  - *Temporal locality:* recently referenced items are likely to be referenced in the near future.
  - *Spatial locality:* items with nearby addresses tend to be referenced close together in time.

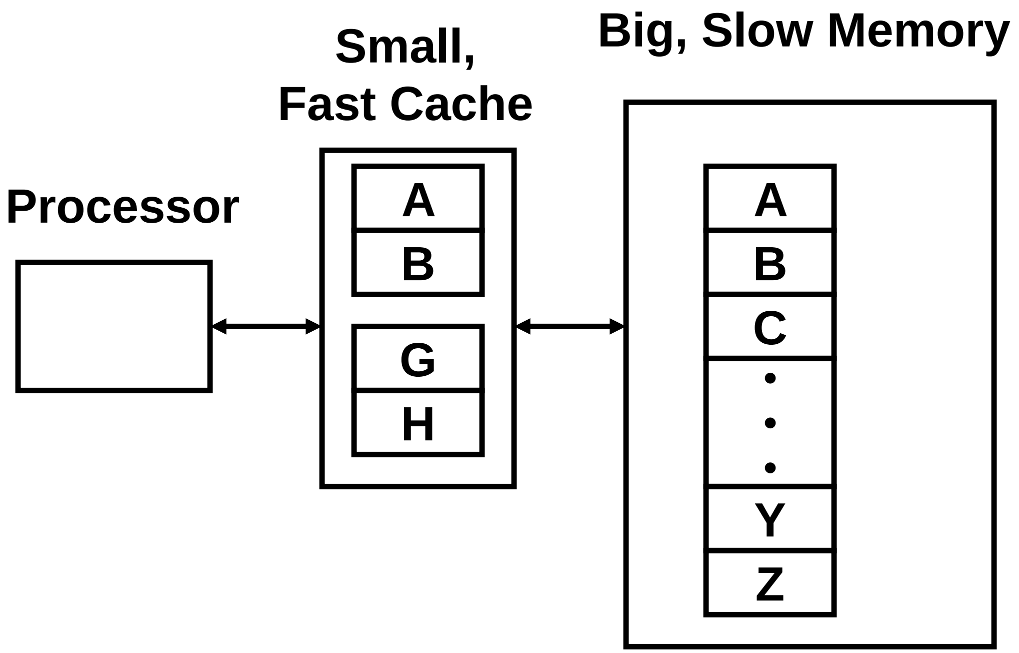
```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
*v = sum;
```

Locality in Example:

- **Data**
  - Reference array elements in succession (spatial)
- **Instructions**
  - Reference instructions in sequence (spatial)
  - Cycle through loop repeatedly (temporal)

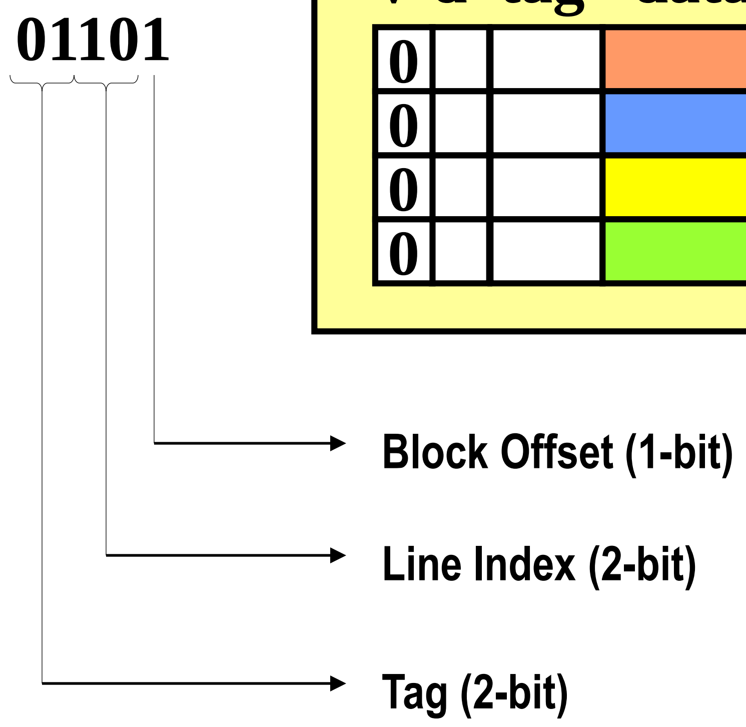
Caching: The Basic Idea

- Main Memory
  - Stores words
    - A–Z in example
- Cache
  - Stores subset of the words
    - 4 in example
  - Organized in lines
    - Multiple words
    - To exploit spatial locality
- Access
  - Word must be in cache for processor to access



Direct-mapped cache

Address



Cache

V	d	tag	data	
0			🟧	🟧
0			🟦	🟦
0			🟨	🟨
0			🟩	🟩

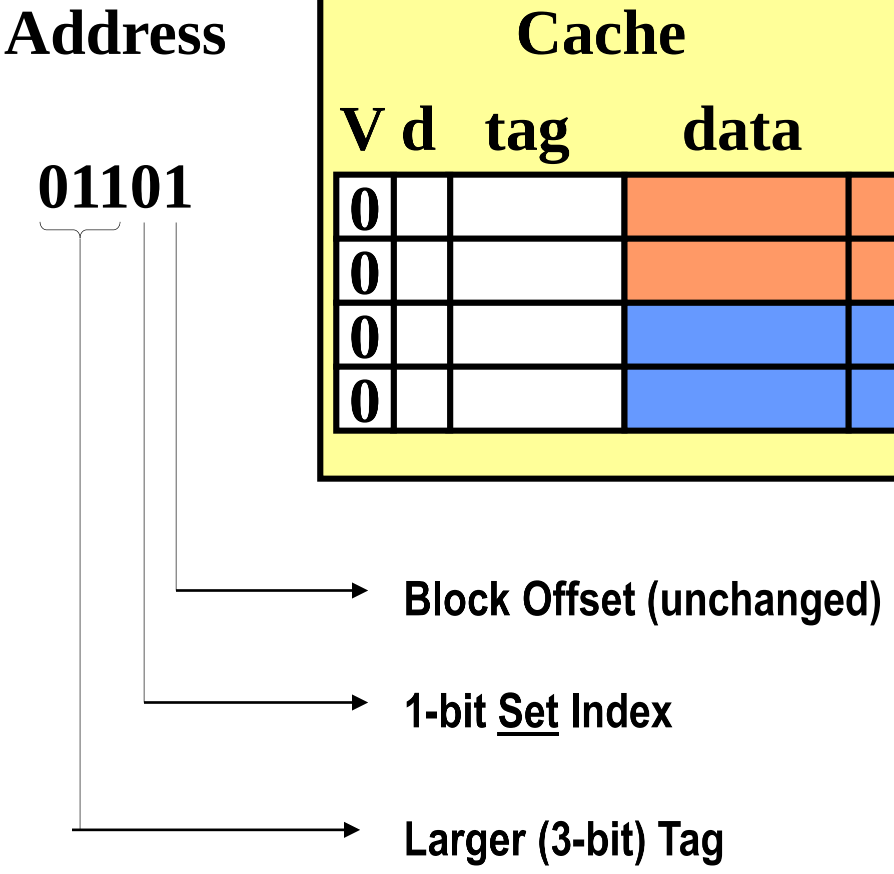
Memory

00000	78	23
00010	29	218
00100	120	10
00110	123	44
01000	71	16
01010	150	141
01100	162	28
01110	173	214
10000	18	33
10010	21	98
10100	33	181
10110	28	129
11000	19	119
11010	200	42
11100	210	66
11110	225	74

3-C's

Compulsory Miss:	first reference to memory block
Capacity Miss:	Working set doesn't fit in cache
Conflict Miss:	Working set maps to same cache line

2-way set associative cache



Cache

V	d	tag	data	
0				
0				
0				
0				

Memory

00000	78	23
00010	29	218
00100	120	10
00110	123	44
01000	71	16
01010	150	141
01100	162	28
01110	173	214
10000	18	33
10010	21	98
10100	33	181
10110	28	129
11000	19	119
11010	200	42
11100	210	66
11110	225	74

Impact on the 3C's?

Parameters

- Total cache size
  - (block size × # sets × associativity)
- Associativity (Number of “ways”)
- Block size (bytes per block)
- Number of sets

Cache overview

- Performance Measures
  - Miss rate
    - % of memory refereces which are not found in the cache.
    - A related measure is #misses per 1000 instructions
  - Average memory access time
    - MR*T_{Miss} + (1-MR)* T_{Hit}
      - T_{Hit} & T_{Miss} --Access time for a hit or miss
- But what do we want to measure?
  - Impact on program execution time.
- What are some flaws of using
  - Miss Rate?
  - Ave. Memory Access Time?
  - Program execution time?
  - Misses per 1000 instructions?

Effects of Varying Cache Parameters

- *Total cache size?*
  - Positives:
    - Should decrease miss rate
  - Negatives:
    - May increase hit time
    - Increased area requirements
    - Increased power (mainly static)
      - Interesting paper:
        - » Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, Trevor N. Mudge: *Drowsy Caches: Simple Techniques for Reducing Leakage Power*. ISCA 2002: 148-157

Effects of Varying Cache Parameters

- ***Bigger block size?***
  - Positives:
    - Exploit spatial locality ; reduce compulsory misses
    - Reduce tag overhead (bits)
    - Reduce transfer overhead (address, burst data mode)
  - Negatives:
    - Fewer blocks for given size; increase conflict misses
    - Increase miss transfer time (multi-cycle transfers)
    - Wasted bandwidth for non-spatial data

Effects of Varying Cache Parameters

- *Increasing associativity*
  - Positives:
    - Reduces conflict misses
    - Low-assoc cache can have pathological behavior (very high miss)
  - Negatives:
    - Increased hit time
    - More hardware requirements (comparators, muxes, bigger tags)
    - Minimal improvements past 4- or 8- way.

Effects of Varying Cache Parameters

- Replacement Strategy: (for associative caches)
  - LRU: intuitive; difficult to implement with high assoc; worst case performance can occur (N+1 element array)
  - Random: Pseudo-random easy to implement; performance close to LRU for high associativity
  - Optimal: replace block that has next reference farthest in the future; hard to implement (need to see the future) ☺

Effects of Varying Cache Parameters

- Write Policy: How to deal with write misses?
  - *Write-through / no-allocate*
    - Total traffic? Read misses × block size + writes
    - Common for L1 caches back by L2 (esp. on-chip)
  - *Write-back / write-allocate*
    - Needs a dirty bit to determine whether cache data differs
    - Total traffic? (read misses + write misses) × block size +
      dirty-block-evictions × block size
    - Common for L2 caches (memory bandwidth limited)
  - *Variation: Write validate*
    - Write-allocate without fetch-on-write
    - Needs sub-block cache with valid bits for each word/byte

4 Hierarchy questions

- Where can a block be placed?
- How do you find a block (and know you’ve found it)?
- Which block should be replaced on a miss?
- What happens on a write?

So from here…

- We need to think in terms of both the hierarchy questions as well as performance.
  - We often will use Average Access Time as a predictor of the impact on execution time. But we will try to keep in mind they may not be the same thing!
- Even all these questions don't get at everything!

Set Associative as a change from Direct Mapped

- Impact of being more associative?
  - MR? T_{Miss}? T_{Hit}?
- Hierarchy questions:
  - Where can a block be placed?
  - How do you find a block (and know you've found it)?
  - Which block should be replaced on a miss?
  - What happens on a write?

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.
- Impact:
  - Impact of being more associative?
    - MR? T_{Miss}? T_{Hit}?
  - Hierarchy questions:
    - Where can a block be placed?
    - How do you find a block (and know you've found it)?
    - Which block should be replaced on a miss?
    - What happens on a write?

Skew cache

- Idea:
  - As hash cache but a different and *independent* hashing function is used for each way.
- Impact:
  - Impact of being more associative?
    - MR? T_{Miss}? T_{Hit}?
  - Hierarchy questions:
    - Where can a block be placed?
    - How do you find a block (and know you've found it)?
    - Which block should be replaced on a miss?
    - What happens on a write?

Victim cache

- Idea:
  - A small fully-associative cache (4-8 lines typically) that is accessed in parallel with the main cache. This victim cache is managed as if it were an L2 cache (even though it is as fast as the main L1 cache).
- Impact:
  - Impact of being more associative?
    - MR? T_{Miss}? T_{Hit}?
  - Hierarchy questions:
    - Where can a block be placed?
    - How do you find a block (and know you've found it)?
    - Which block should be replaced on a miss?
    - What happens on a write?

WHAT’S HARD ABOUT IMPLEMENTING CACHES?

So…

- What's hard about caches?
  - And in particular, for the project?
- There are a lot of implementation details that get tricky
  - LRU is tricky to do above 2 way
  - Critical word first
    - less important for the project, but a real issue.
  - Making a non-blocking cache is tricky
  - Writes are tricky
  - Dealing with load/store dependencies
  - Multi-processor issues

Critical Word First

- Idea:
  - For caches where the line size is greater than the word size, send the word which causes the miss first
- Why?
  - As always, the processor is what matters.
  - We are getting extra data (cache line) into the cache to handle *future* requests
  - But we want to get the current request handled as quickly as possible.
- What's hard?
  - Memory system needs to support this.
  - We need to take an action while the data is still arriving.

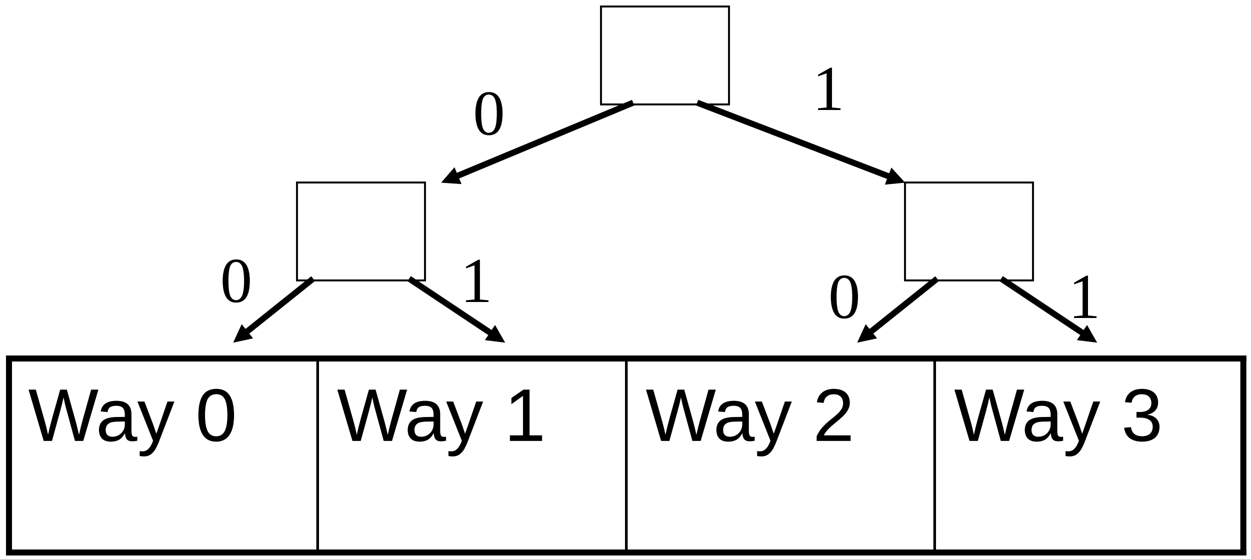
LRU is difficult

- We want to keep track of the exact order things have been used in the set.
  - So if I have 4 things, I want to know the MRU, the next MRU, etc.
- I could maintain a stack-like structure
  - But that's a lot of data movement.
- I could maintain a counter for each line
  - But that's a lot of work and area (why area?)
- No good way to track LRU?
  - Something smart?

Pseudo-LRU replacement

- # of bits needed to maintain order among N items?
- So for N=16 we need: _45___ bits.
- Any better ideas?

Psuedo LRU



General theme:

On a hit or replacement, switch all the bits to point away from you.

Replace the one pointed to.

Handling Multiple Outstanding Accesses

- Question: If the processor can generate multiple cache accesses, can the later accesses be handled while a previous miss is outstanding?
- Goal: Enable cache access when there is a pending miss
- Goal: Enable multiple misses in parallel
  - Memory-level parallelism (MLP)
- Solution: Non-blocking or lockup-free caches
  - Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," ISCA 1981.

This section from Dr. Onur Mutlu

Handling Multiple Outstanding Accesses

- Idea: Keep track of the status/data of misses that are being handled in Miss Status Handling Registers (MSHRs)
  - A cache access checks MSHRs to see if a miss to the same block is already *pending.*
    - If pending, a new request is not generated
    - If pending and the needed data available, data forwarded to later load
  - Requires buffering of outstanding miss requests

Miss Status Handling Register

- Also called “miss buffer”
- Keeps track of
  - Outstanding cache misses
  - Pending load/store accesses that refer to the missing cache block
- Fields of a single MSHR entry
  - Valid bit
  - Cache block address (to match incoming accesses)
  - Control/status bits (prefetch, issued to memory, which subblocks have arrived, etc)
  - Data for each subblock
  - For each pending load/store
    - Valid, type, data size, byte in block, destination register or store buffer entry address

MSHR Operation

- On a cache miss:
  - Search MSHRs for a pending access to the same block
    - Found: Allocate a load/store entry in the same MSHR entry
    - Not found: Allocate a new MSHR
    - No free entry: stall
- When a subblock returns from the next level in memory
  - Check which loads/stores waiting for it
    - Forward data to the load/store unit
    - De-allocate load/store entry in the MSHR entry
  - Write subblock in cache or MSHR
  - If last subblock, de-allaocate MSHR (after writing the block in cache)

Non-Blocking Cache Implementation

- When to access the MSHRs?
  - In parallel with the cache?
  - After cache access is complete?

- MSHRs need not be on the critical path of hit requests
  - Which one below is the common case?
    - Cache miss, MSHR hit
    - Cache hit

Miss Status Handling Register Entry

1	27	1
Valid	Block Address	Issued

1	3	5	5	
Valid	Type	Block Offset	Destination	Load/store 0
Valid	Type	Block Offset	Destination	Load/store 1
Valid	Type	Block Offset	Destination	Load/store 2
Valid	Type	Block Offset	Destination	Load/store 3

So…

- What's hard about caches?
  - And in particular, for the project?
- There are a lot of implementation details that get tricky
  - LRU is tricky to do above 2 way
  - Critical word first
    - less important for the project, but a real issue.
  - Making a non-blocking cache is tricky
  - Writes are tricky
  - Dealing with load/store dependencies
  - Multi-processor issues

3C’s model

- Break cache misses into three categories
  - Compulsory miss
  - Capacity miss
  - Conflict miss
- Compulsory
  - The block in question had never been accessed before.
- Capacity
  - A fully-associative cache of the same size would also have missed this access given the same reference stream.
- Conflict
  - That fully-associative cache would have gotten a hit.

3C’s example

- Consider the “stream” of blocks 0, 1, 2, 0, 2, 1
  - Given a direct-mapped cache with 2 lines, which would hit, which would miss?
  - Classify each type of miss.

3C’s – sum-up.

- What’s the point?
  - Well, if you can figure out what kind of misses you are getting, you might be able to figure out how to solve the problem.
    - How would you “solve” each type?
- What are the problems?

Reference stream

- A memory ***reference stream*** is an *n*-tuple of addresses which corresponds to *n* ordered memory accesses.
  - A program which accesses memory locations 4, 8 and then 4 again would be represented as (4,8,4).

Locality of reference

- The reason small caches get such a good hit-rate is that the memory reference steam is not random.
  - A given reference tends to be to an address that was used recently. This is called ***temporal locality***.
  - Or it may be to an address that is near another address that was used recently. This is called ***spatial locality.***
- Therefore, keeping recently accessed blocks in the cache can result in a remarkable number of hits.
  - But there is no known way to quantify the amount of locality in the reference stream.

Stack distance – A measure of locality

- Consider the reference stream

  (0, 8, 4, 4, 12, 16, 32, 4)

- If the cache line size is 8 bytes, the block reference stream is

  (0, 1, 0, 0, 1, 2, 4, 0)

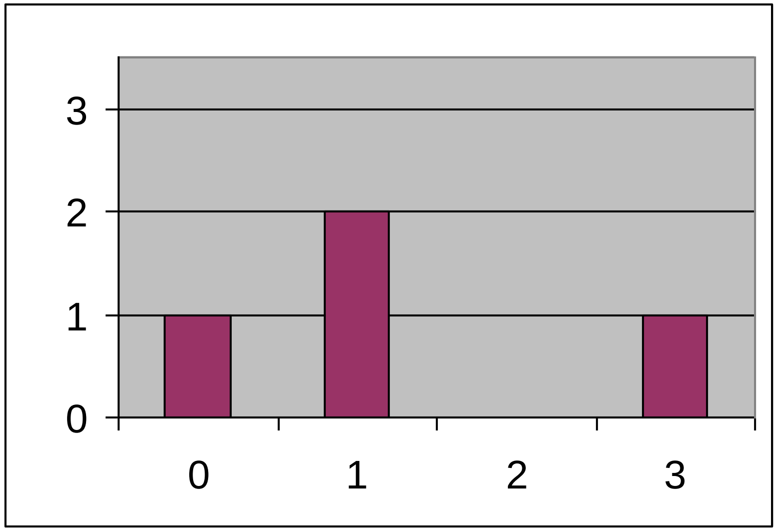
- Now define the ***stack distance*** of a reference to be the number of unique block addresses between the current reference and the previous reference to the same block number. In this case

  (∞, ∞, 1, 0, 1, ∞, ∞, 3)

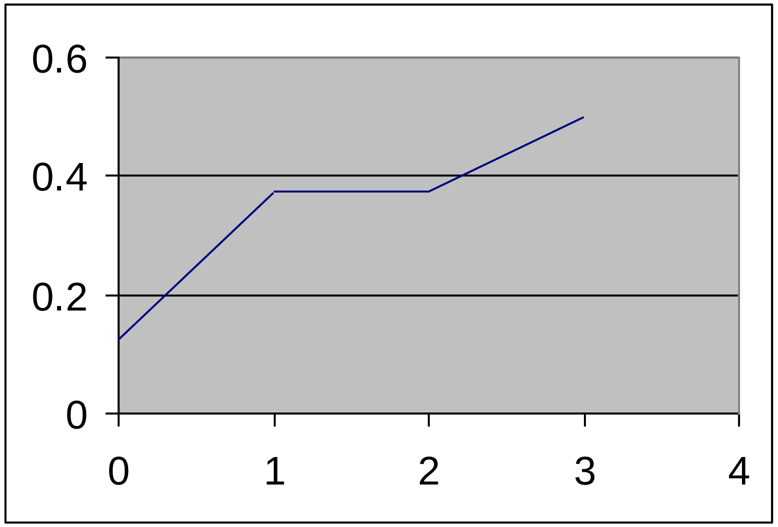
Stack distance – A measure of locality (2)

Memory reference stream	0	8	4	4	12	16	32	4
Block reference stream	0	1	0	0	1	2	4	0
Stack distance	∞	∞	1	0	1	∞	∞	3

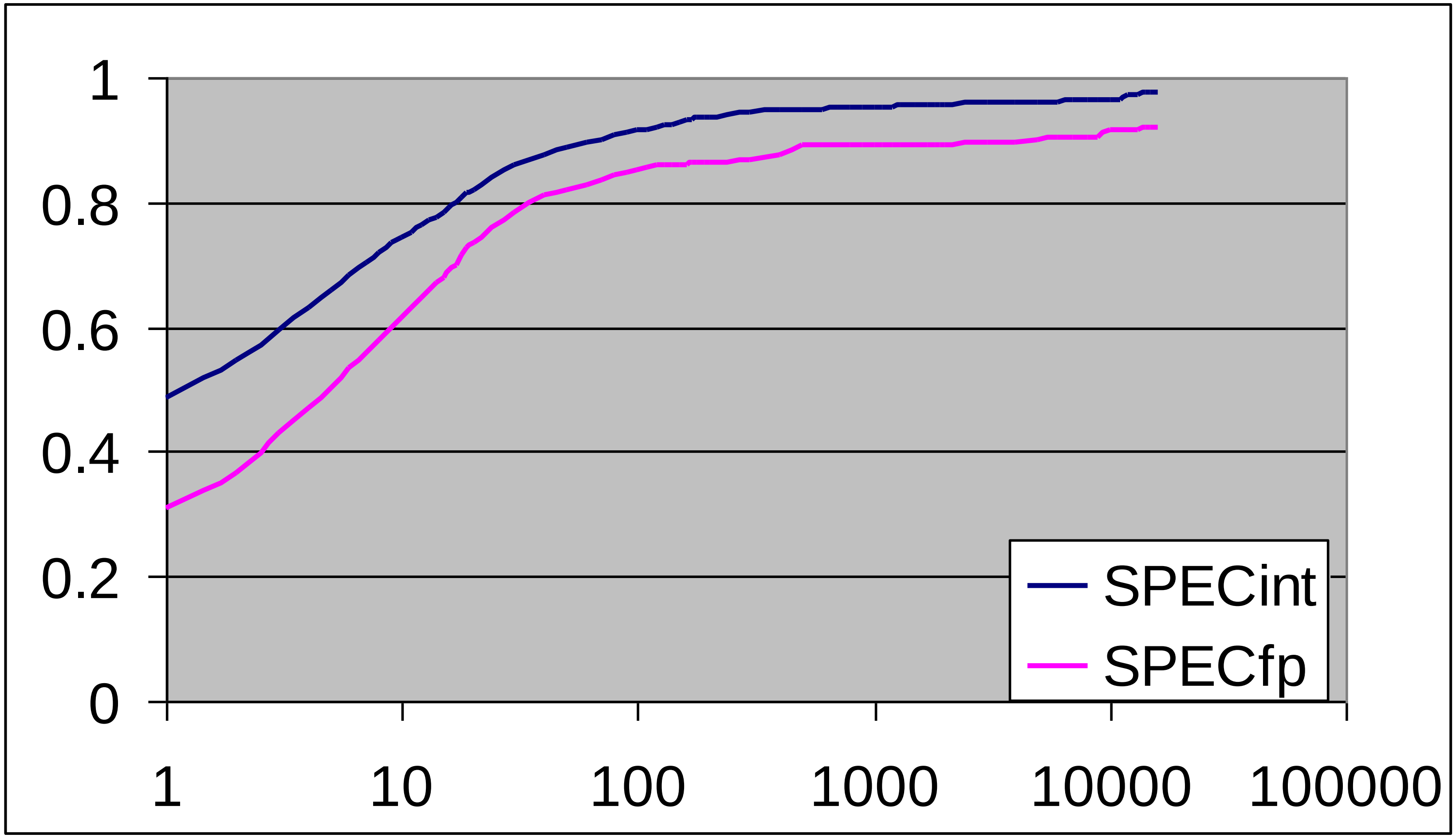
Number non-infinite accesses



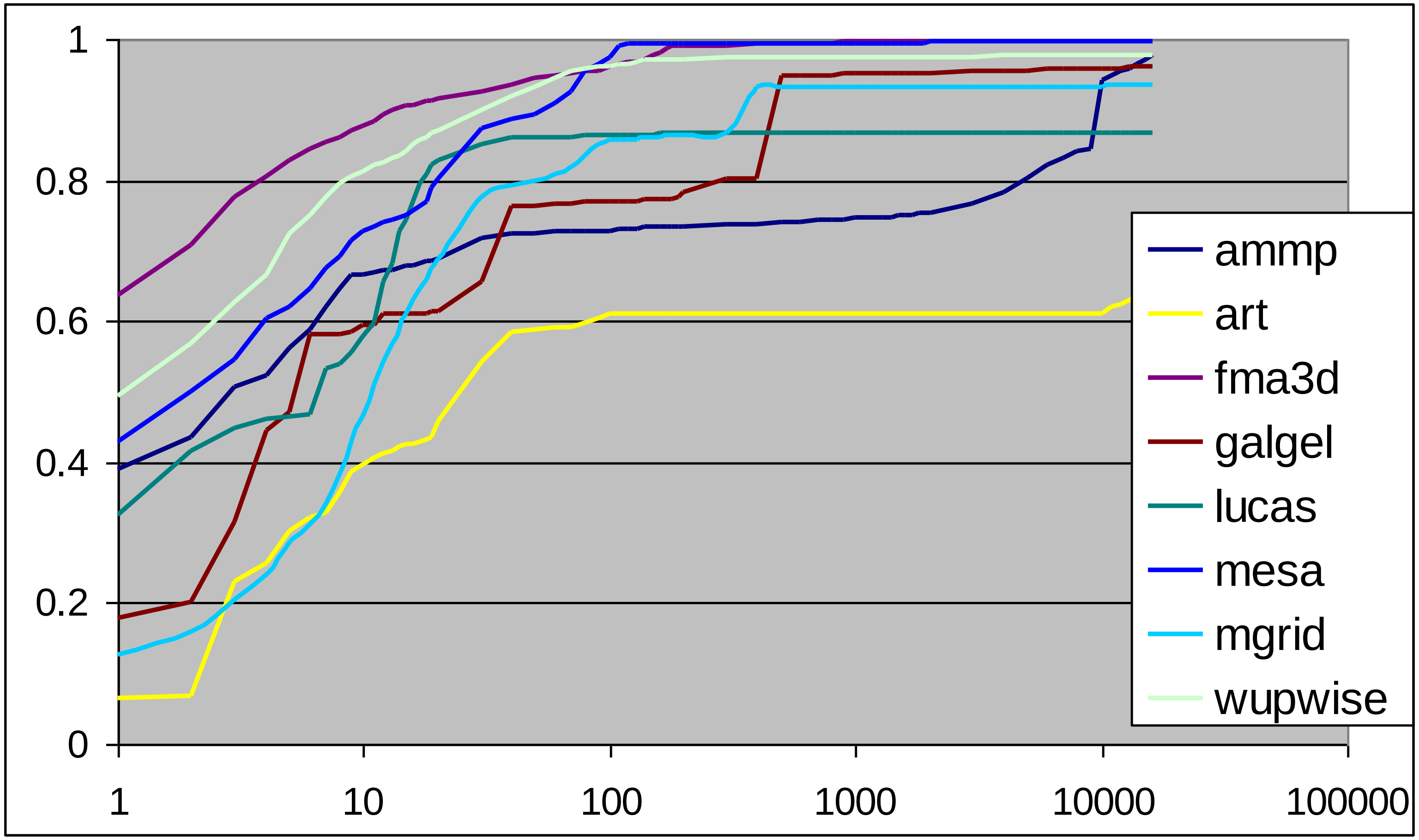
Cumulative stack distance



Stack distances of the SPEC benchmarks – cumulative



Stack distances of selected SPECfp benchmarks



Why is this interesting?

- It is possible to distinguish locality from conflict.
  - Pure miss-rate data of set-associative caches depends upon both the locality of the reference stream and the conflict in that stream.
- It is possible to make qualitative statements about locality
  - For example, SPECint has a higher degree of locality than SPECfp.

Fully-associative caches

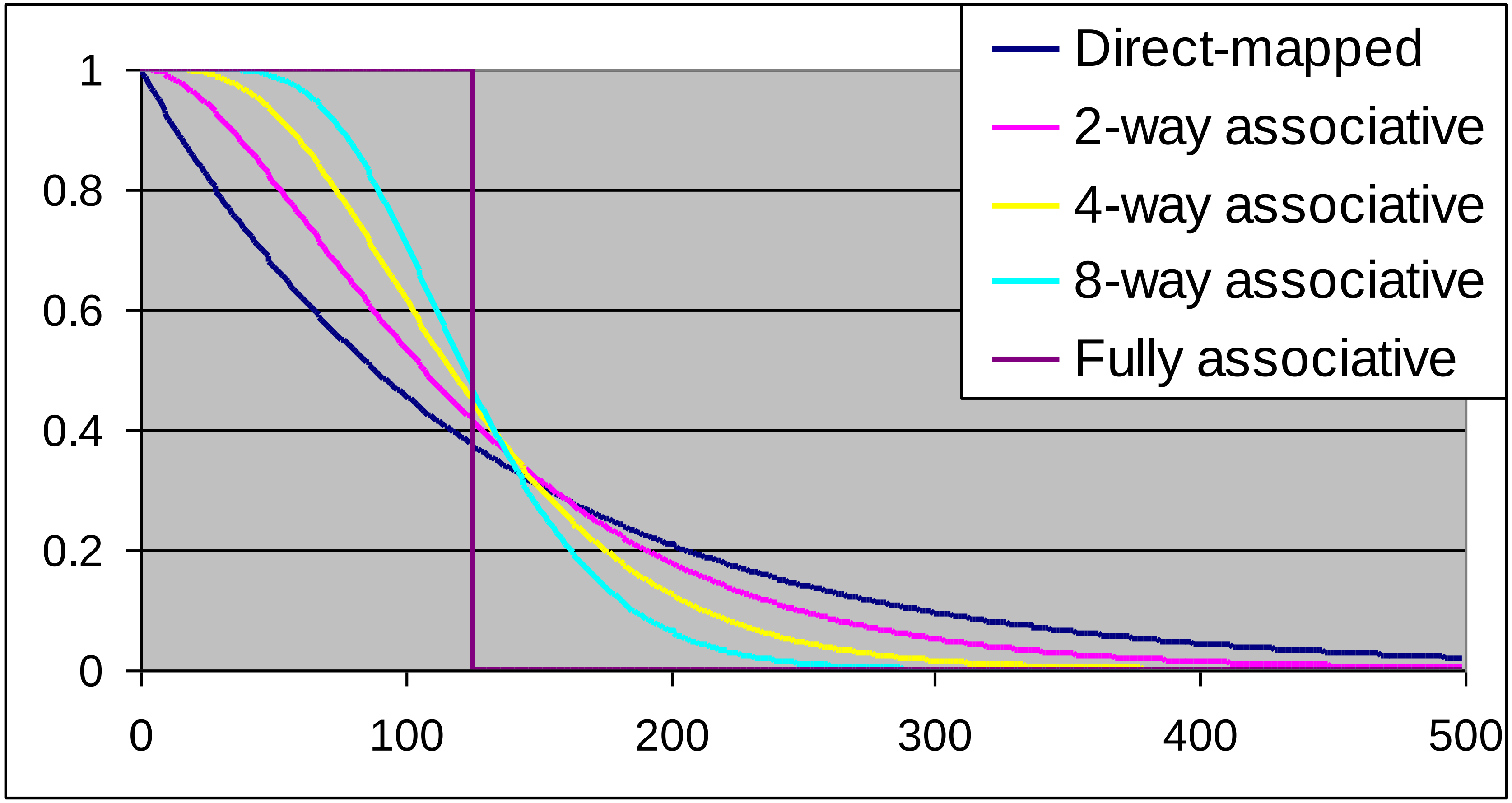
- A fully-associative LRU cache consisting of *n* lines will get a hit on any memory reference access with a stack distance of *n-1* or less.
  - Fully associative caches of size *n* store the last *n* uniquely accessed blocks.
  - This means the locality curves are simply a graph of the hit rate on a fully associative cache of size *n-1.*

Direct-mapped caches

- Consider a direct-mapped cache of *n* cache lines and the block reference stream (***x, y, x***).
  - The second reference to ***x*** will be a hit unless ***y*** is mapped to the same cache line as x.
  - If ***x*** and ***y*** are independent, the odds ***x*** being a hit is (n-1)/n
- In general if there are ***m*** unique accesses between the two references to ***x***, the odds of a hit are:

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

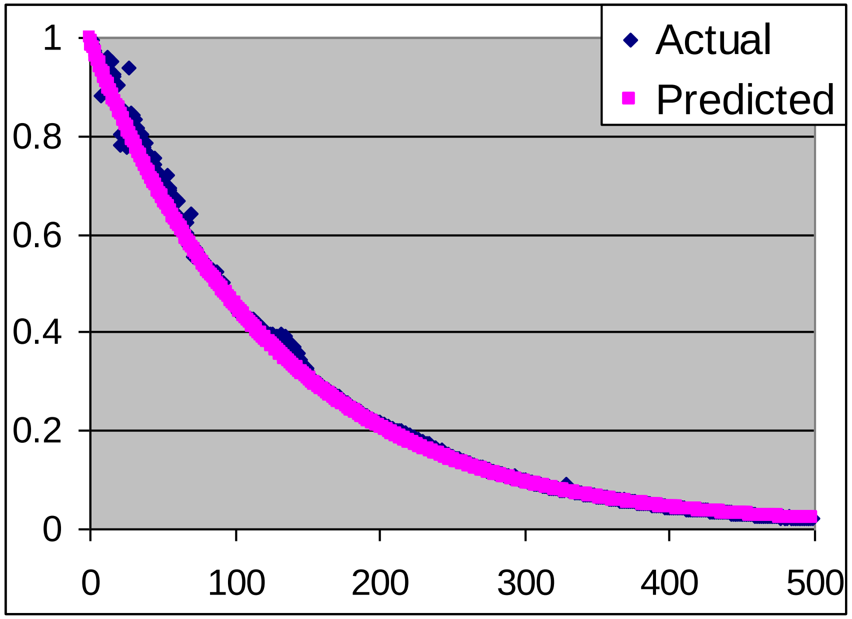
Expected hit rate at a given stack distance for a 128-line cache

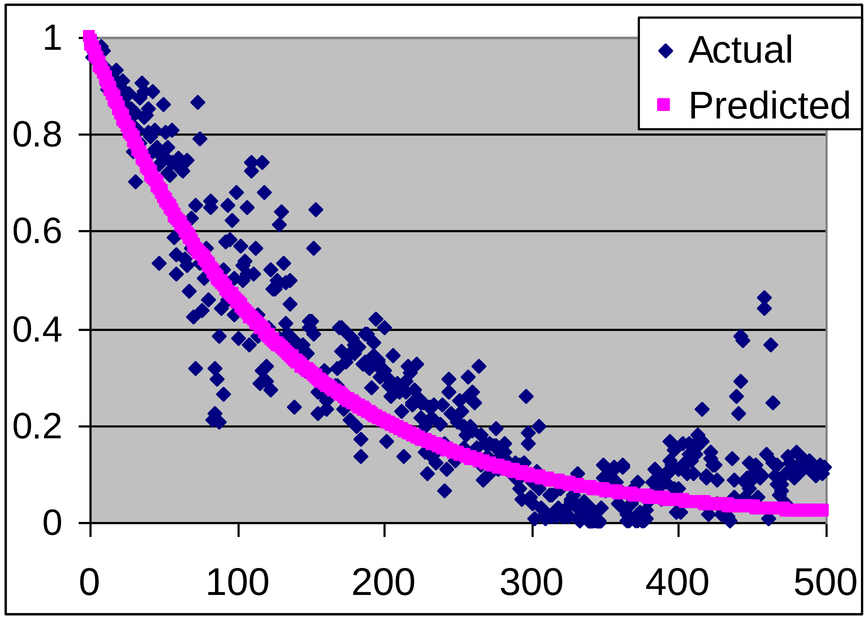


Verification – direct mapped 128-entry cache

SPECint

SPECfp





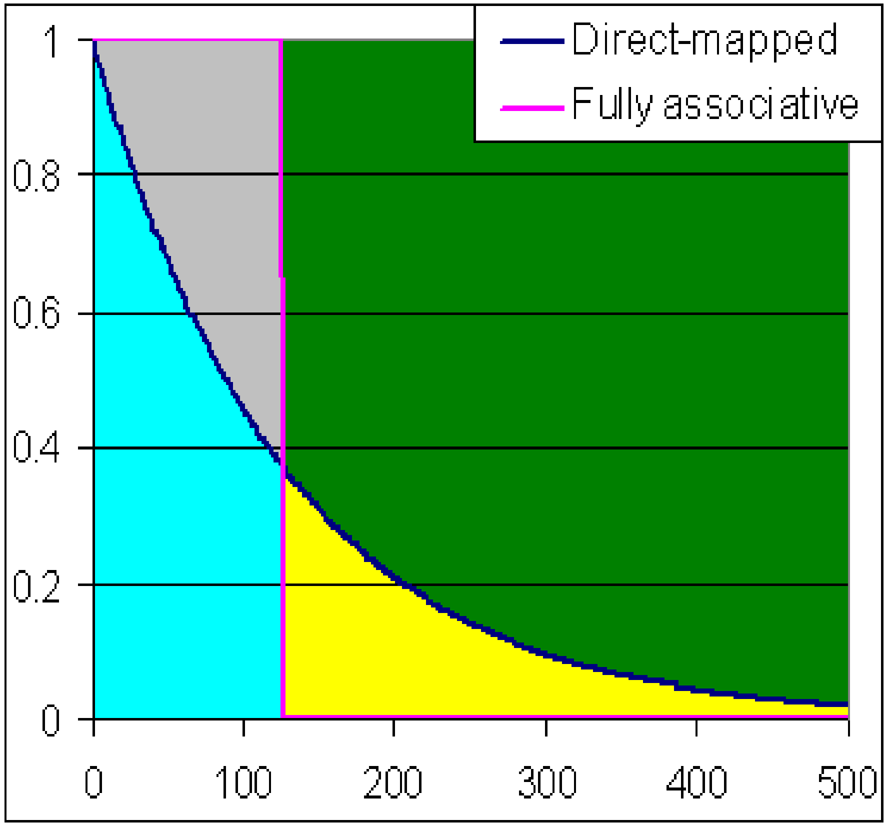
Why is this interesting?

- 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.

The 3C's model

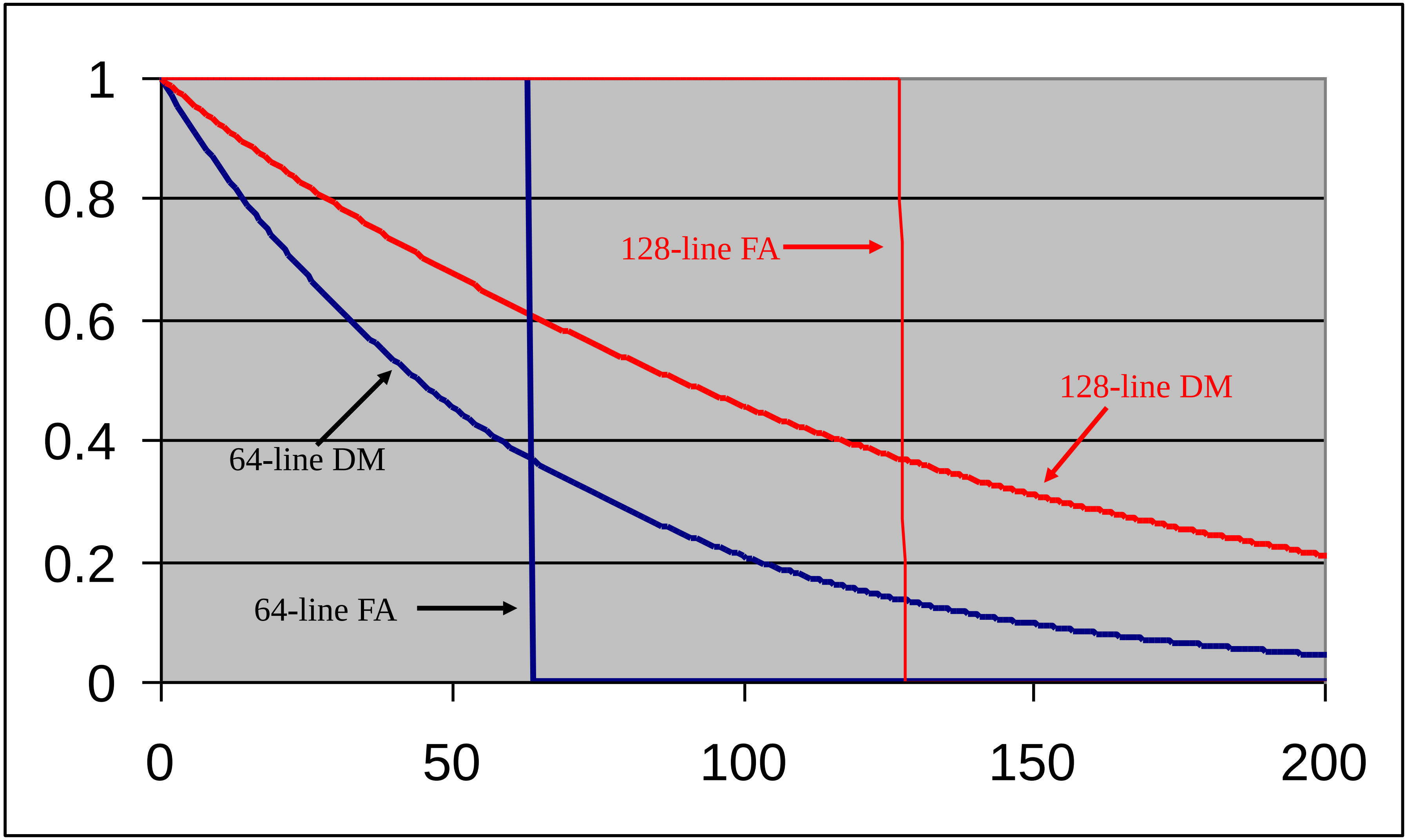
The 3'Cs model describes misses as conflict, capacity or compulsory by comparing a direct-mapped cache to a fully-associative cache.

- 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"?



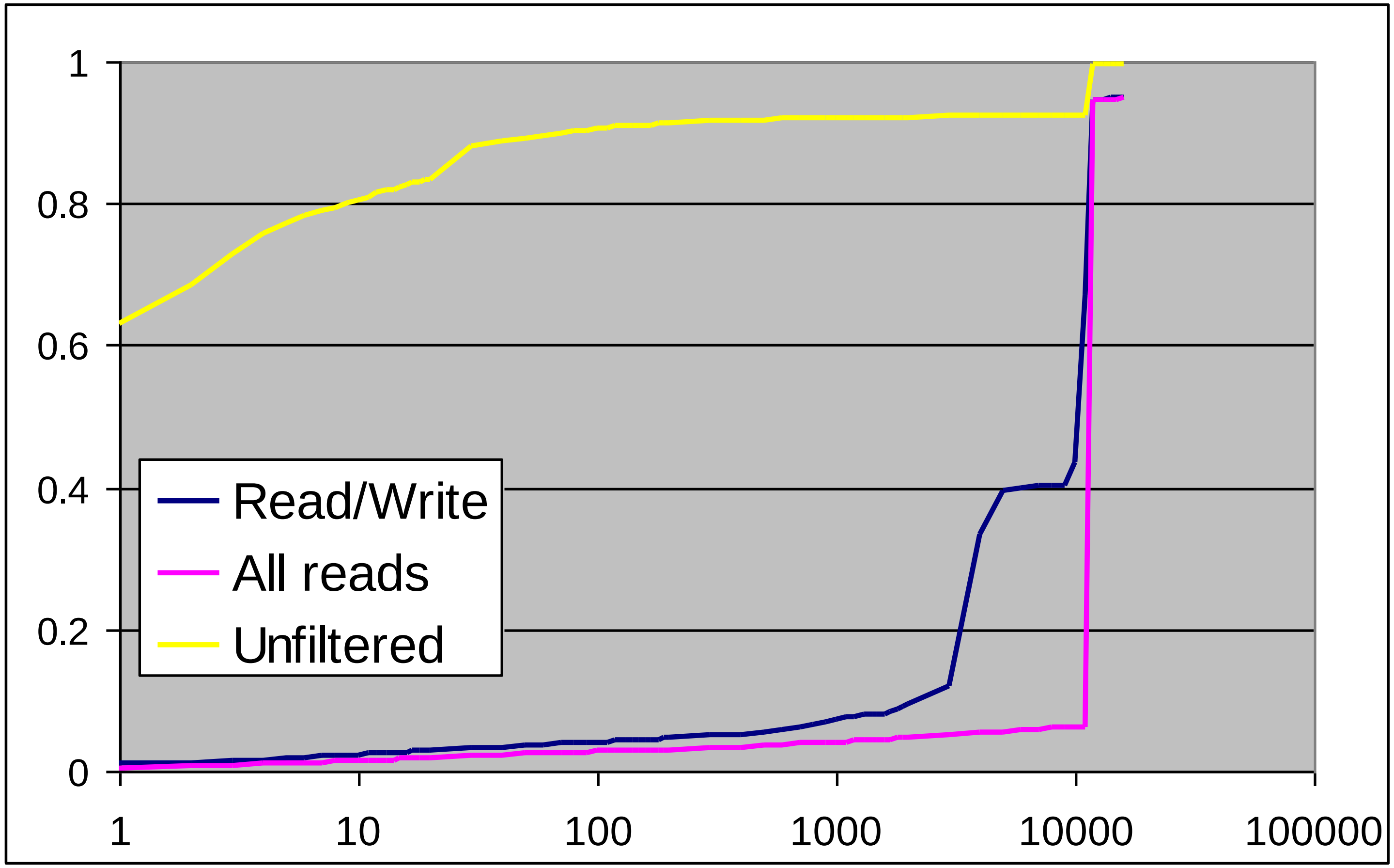
The 3C's model is much more limited.

It cannot distinguish between *expected* conflict and *excessive* conflict.


1
0.8
0.6
0.4
0.2
0
0
50
100
150
200
128-line FA
128-line DM
64-line DM
64-line FA

The filtering of locality (1/2)

gcc after a 64KB direct-mapped cache



The filtering of locality (2/2)

- Notice that there fewer references at the lowest stack distances than at the highest.
  - Yet it is at the lowest stack distances where highly-associative caches concentrate their power.
  - The locality seen by the L2 cache is *fundamentally* different than that seen by the L1 cache
    - A large enough cache can make this effect go away…

Measuring non-random conflict (1/2)

- Combining the cache and locality models makes it possible to predict a hit rate.

Benchmark	**Average predicted hit rate**	**Average actual hit rate**
SPECint	**90.81%**	**91.38%**
SPECfp	**83.19%**	**83.84%**

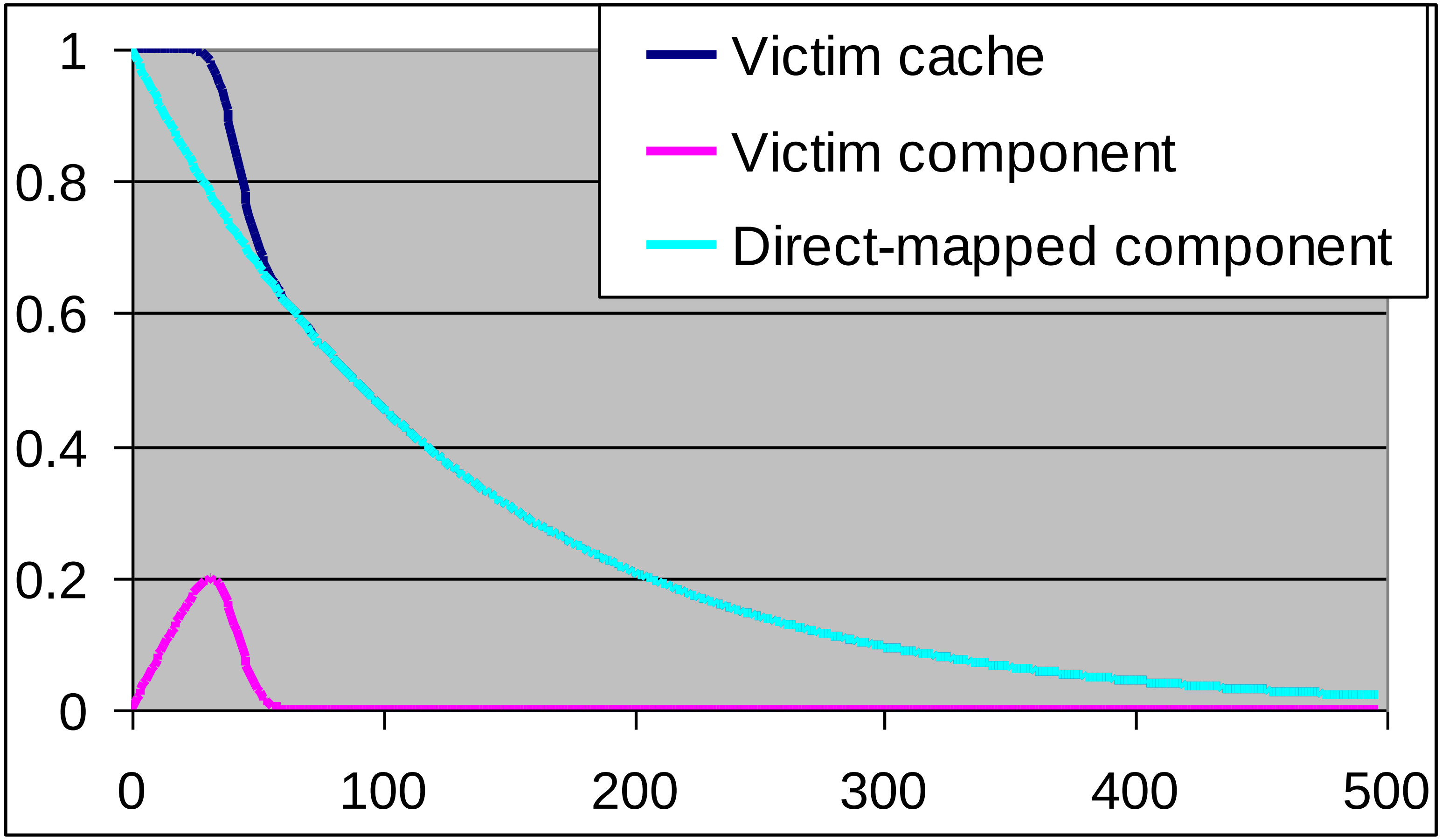
  – On these reference streams the cache tends to do better than predicted.
    - Although some benchmarks, like mgrid, see significantly worse performance.

Measuring non-random conflict (2/2)

- A more advanced technique, using a hash-cache, allows us to roughly quantify the amount of *excessive* conflict and *scant* conflict.
  - This can be useful when deciding if a hash cache is appropriate.
  - It is also useful to provide feedback to the compiler about its data-layout choices.
- Other compilers (gcc for example) tend to have a higher degree of excessive conflict.
  - So this technique may also be able to tell us something about compliers.

Understanding non-standard caches

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



Some review

- Consider the access pattern A, B, C, A. Assume the three accesses are all independently randomly placed with uniform probability
  - In a direct-mapped cache with 8 lines, what is the probability of a miss?
  - A two-way associative cache with 4 lines?
  - A victim cache of 1 one backing up a direct-mapped cache of 4 lines?
  - A skew cache with 8 lines?
- What is bogus about the above assumptions?

VIPT caches

- (Done on board)