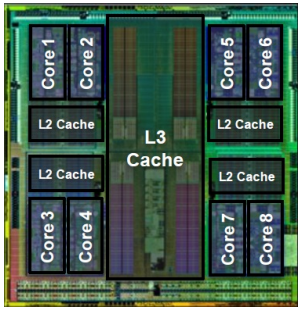
Lecture 5
Low-Miss-Ratio Caches

Fall 2016

Pejman Lotfi-Kamran

AMD FX



Adapted from slides originally developed by Profs. Falsafi, Hill, Hoe, Lipasti, Shen, Smith, Sohi, and Vijaykumar of Carnegie Mellon University, EPFL, Purdue University, and University of Wisconsin.

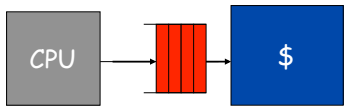
Where Are We?

Fr	Sa	Su	Mo	Tu
	27-Shahrivar		29-Shahrivar	
	3-Mehr		5-Mehr	
	10-Mehr		12-Mehr	
	17-Mehr		19-Mehr	
	24-Mehr		26-Mehr	
	1-Aban		3-Aban	
	8-Aban		10-Aban	
	15-Aban		17-Aban	
	22-Aban		24-Aban	
	29-Aban		1-Azar	
	6-Azar		8-Azar	
	13-Azar		15-Azar	
	20-Azar		22-Azar	
	27-Azar		29-Azar	
	4-Dey		6-Dey	

This Lecture
- Low-Miss-Ratio Caches

Next Lecture:
- High-B/W Caches

Write Buffers



Buffer CPU writes
- allows reads to proceed
- stall only when full
- data dependence?
  - What happens on dependent loads/stores?

Write Buffers (Cont.)

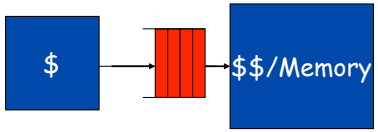
Write Policy	Write Alloc	Hit/Miss	Write Buffer writes to
Back	Yes	Both	Cache
Back	No	Hit	Cache
Back	No	Miss	Memory
Through	Yes	Both	Both
Through	No	Hit	Both
Through	No	Miss	Memory

Write Buffers (Cont.)

Design issues:

- Design for bursts
- Coalesce adjacent writes?
- Sixteen entries is typical

Writeback Buffers



Between write-back cache and next level

1. Move replaced, dirty blocks to buffer
2. Read new line
3. Move replaced data to memory

Usually need 8 write-back buffer entries

Mark Hill's DM vs. SA: "Bigger & Dumber is Better"

$t_{avg} = t_{hit}$ + miss ratio x t_{miss}

- comparable DM and SA caches with same t_{miss}
- but, associativity that minimizes t_{avg} is often smaller than associativity that minimizes miss ratio

remember:

$$diff(t_{cache}) = t_{cache}(SA) - t_{cache}(DM) \geq 0$$

$$diff(miss) = miss(SA) - miss(DM) \leq 0$$

e.g.,

assuming $diff(t_{cache}) = 0$ => SA better

assuming $diff(miss) = -1\%$, $t_{miss} = 20$

$\Rightarrow$ if $diff(t_{cache}) > 0.2$ cycle then SA loses

"Harvard" vs. "Princeton"

Unified *(sometimes known as Princeton)*

- less costly, dynamic response, handles writes to instructions

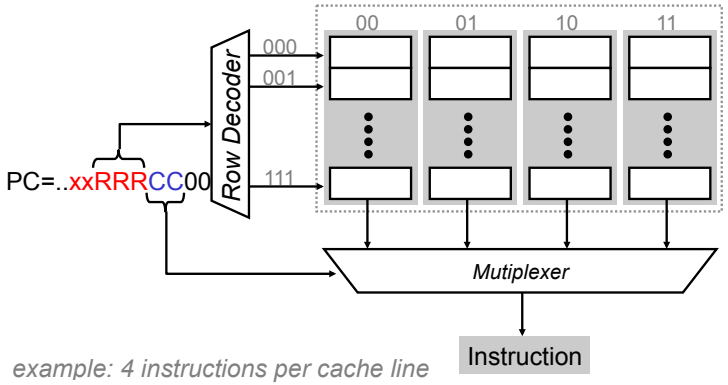
Split I and D *(sometimes known as Harvard)*

- most of the time code and data don't mix
- 2x bandwidth, place close to I/D ports
- can customize size (I-footprint generally smaller than d-footprint), no interference between I/D
- self-modifying code can cause "coherence" problems

Caches should be split for frequent simultaneous I & D access

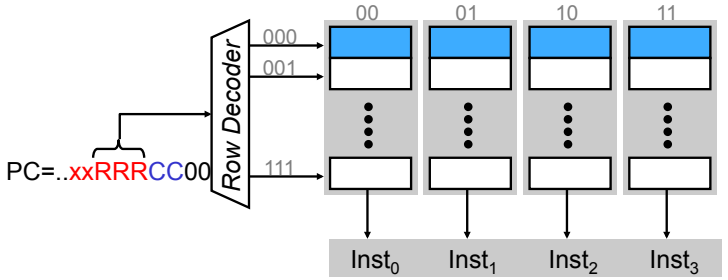
- no longer a question in "high-performance" on-chip L-1 caches

L1 Instruction Cache Issues

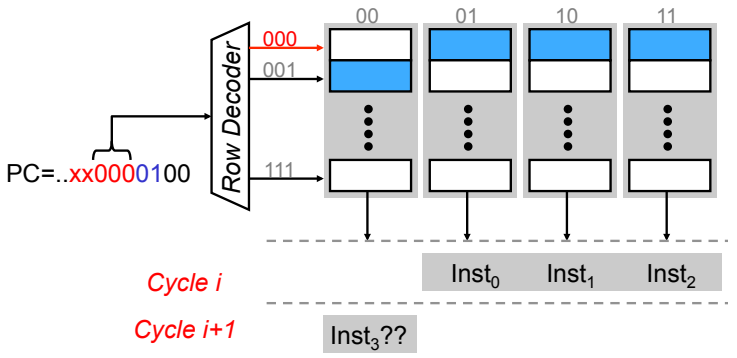


example: 4 instructions per cache line

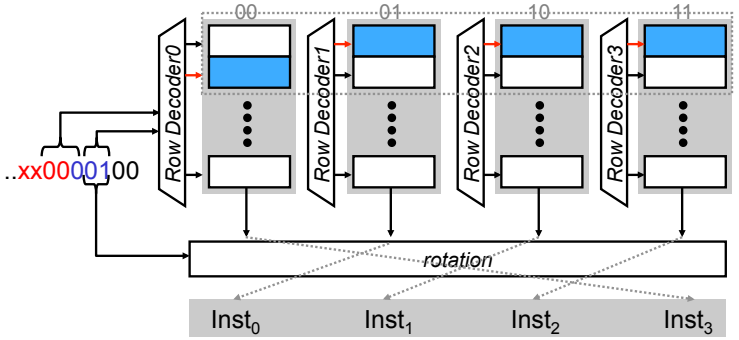
Spatial Locality and Fetch Bandwidth



Fetch Group Misalignment



Auto Alignment



- A block frame physically spans multiple SRAM lines *(**What is a block frame?)*
- Do you always get the maximum number of instructions?

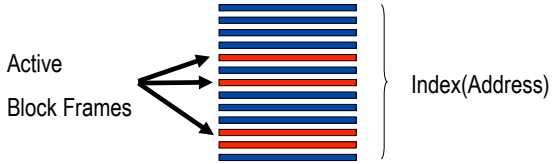
How to Reduce Conflict Misses?

Why do conflict misses occur?

(assuming no capacity concerns)

Uneven distribution of active block frame into sets

- active frames with the same index will map to the same set



What causes uneven distribution?

probabilistic vs programmatic (e.g. multi-dim. array traversal)

How do we redistribute blocks into block frames?

Use Hash Functions

Current mapping function only uses index bits

- conflict happens for blocks with identical index bits

Can we use other parts of the address (tag bits) to distinguish?

i.e. use more than n-bit to index into 2^n-line SRAM

- For $f_{index}(addr0) = f_{index}(addr1)$ can chose f_{hash} so that
- $f_{hash}(addr0) != f_{hash}(addr1)$

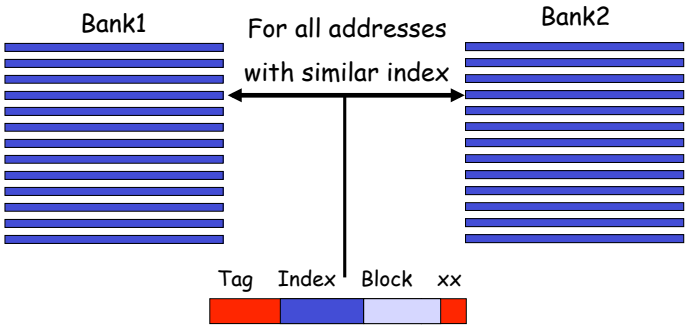
Cons:

1. What is a good hash function? Do we need program specific hash functions?
2. How expensive is it to critical path?

- Pros:

1. Can lead to better effective use of cache

Regular Set-Associative Cache



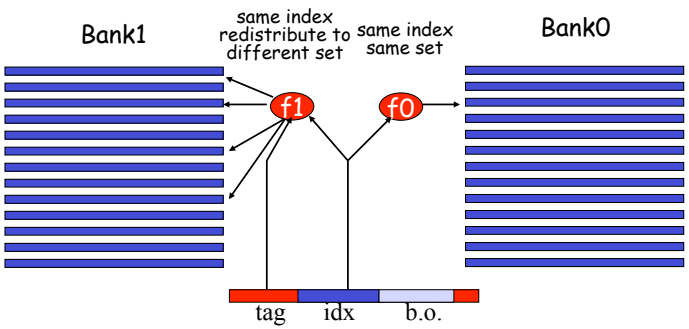
Seznec's Skewed-Associative Cache

For 4-way skewed-associative cache consider:

bank0: A1 xor A2
bank1: shuffle(A1) xor A2
bank2: shuffle(shuffle(A1)) xor A2
bank3: shuffle(shuffle(shuffle(A1))) xor A2

Implementation only adds xor's to cache access path

Seznec's Skewed-Associative Cache

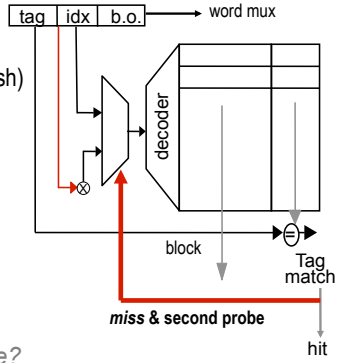


Can get better utilization with less assoc?
average case? worst case?

Column-Associative Cache

DM cache structure

- Look up regularly
- If miss, xor address (i.e., simple hash)
- Look up again
- If found, swap
- If miss, fetch from next level



What is wrong with this Cache?

Pseudo-Associativity: e.g MIPSR10K 2-way L2

Classic associativity is too expensive for external SRAMS (chip-count and routing)

N-way associativity is a placement policy

- it says an address could be mapped to N different locations in the cache
- it doesn't say all look up needs to be parallel

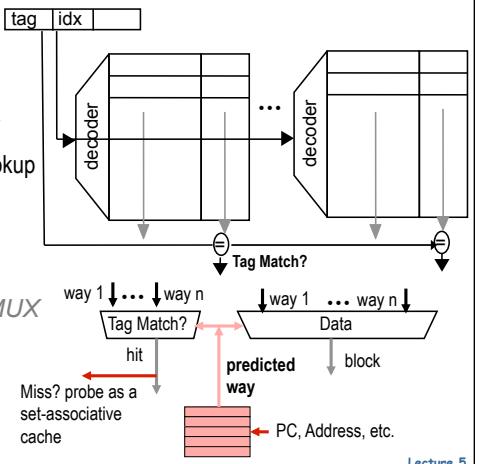
Pseudo N-way Associativity:

- Given a direct-mapped array with K cache blocks
- Implement K/N sets
- Given address Addr, sequentially look up:
  - {0,Addr[lg(K/N)-1: 0]}, {1,Addr[lg(K/N)-1: 0]}, …
    {N-1,Addr[lg(K/N)-1: 0]}
- Optimization: record the most recently used way (MRU) to look up first

Way-Predicting Set-Associative Cache

Set-associative structure

- Predict way
- Look up that way only
- If miss, regular SA lookup



Hmm..Still goes thru MUX
How does this help?

Daniel Sanchez' ZCache

One hash function per way (Similar to skewed-assoc caches)

Lookup similar to skewed-assoc caches

- Hit is identical to to skewed-assoc caches

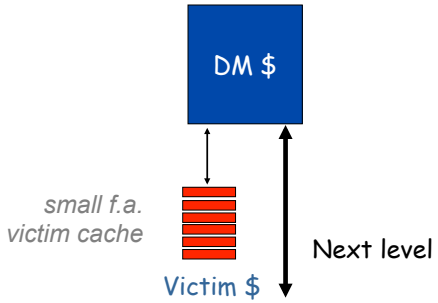
On a miss, more victims are gathered using Cuckoo hashing

- Apply hash functions on victims to gather more victims

Advanced Caches (Cont.)

- Miss Ratio
  - Associativity
  - Prefetching (later)
- Miss Cost
  - Victim caching (related to associativity)
  - Request-word first
  - Subblocking
  - Multi-level caches
  - Lock-up free caches
- Issue bandwidth

Norm Jouppi's Victim Cache



Lines evicted from the direct-mapped cache due to collision is stored into the victim cache

Avoids ping-ponging when the working set contains a few addresses that collides

Norm Jouppi's Victim Cache

Targets conflict misses

Victim cache: a small fully-associative cache capturing "victims"

- victims are conflicts in a set in DM or low-assoc cache
- LRU replacement

A miss in cache + a hit in victim cache

- move line to main cache
- is effectively equal to fast miss handling (or slow hits)

Victim Cache's Performance

Removing conflict misses
- even one entry helps some benchmarks
- helps I-cache more than D-cache

Compared to cache size
- generally, victim cache helps more for smaller caches

Compared to line size
- helps more with larger line size (why?)

Reducing Miss Cost

Assume 8 cycles before delivering 2 words/cycle
- $t_{memory} = t_{access} + B \times t_{transfer} = 8 + B \times 1/2$
- B is the block size in words
- whole block loaded before CPU gets data

Assume requested word first
- CPU gets data before cache loads it in data array
- $t_{memory} = t_{access} + MB \times t_{transfer} = 8 + 2 \times 1/2$
- MB is memory bus width in words

Large Blocks and Subblocking

Large cache blocks can take a long time to refill
- refill cache line *critical word first*
- restart cache access before complete refill

Large cache blocks can waste bus bandwidth if block size is larger than spatial locality
- divide a block into subblocks
- associate separate valid bits for each subblock

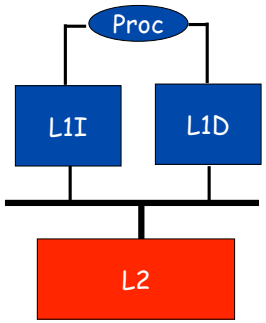
Multi-Level Caches

Processors getting faster w.r.t. main memory
- larger caches to reduce frequency of more costly misses
- but larger caches are too slow for processor

=> gradually reduce cost of misses with a multiple cache levels

$$t_{avg} = t_{hit} + \text{miss ratio} \times t_{miss}$$

Multi-Level Cache Design



different requirements
⇒ different choice of
capacity
block size
associativity

$t_{avg\text{-}L1} = t_{hit\text{-}L1} + \text{miss-ratio}_{L1} \times t_{avg\text{-}L2}$

$t_{avg\text{-}L2} = t_{hit\text{-}L2} + \text{miss-ratio}_{L2} \times t_{memory}$

The Inclusion Property

Multi-level inclusion holds if L2 is always a superset of L1, and so on down the hierarchy. Why?

- if an addr is in L1, then it must be frequently used
- makes L1 design simpler
  don't need separate cases for L2 hit vs. miss
- L2 can handle external coherence checks without L1

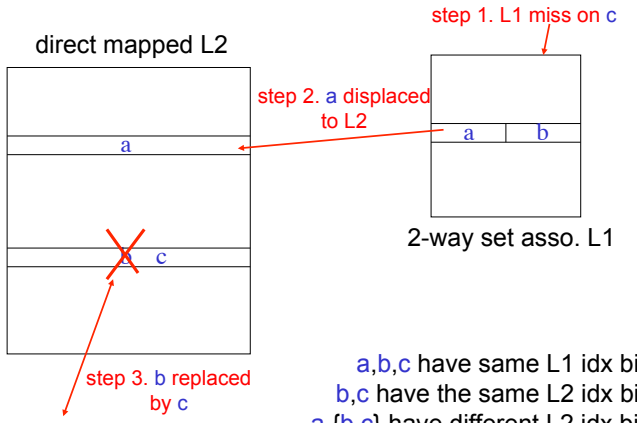
Causes interesting interactions between L1I, L1D, and L2

Inclusion takes effort to maintain

- each L2 block frame needs status bits marking subblocks in cached by L1
  (# of bit = L2 blocksize/L1 blocksize)
- if a L2 block is to be displaced, must first flush out all of its L1-cached subblocks

Why would an L2 block be displaced
if it is still has subblocks in L1?
Consider 1. L1 block size < L2 block size 2. associativity in L1

Possible Inclusion Violation



a,b,c have same L1 idx bits
b,c have the same L2 idx bits
a,{b,c} have different L2 idx bits

Multi-Level Inclusion (Cont.)

Inclusion takes effort to maintain

- make L2 cache have bits or pointers giving L1 contents
- invalidate from L1 before replacing from L2
- number of pointers per L2 block
  - L2 blocksize/L1 blocksize

Interesting interaction between L1I, L1D, and L2

- Example a matrix multiply program for a large matrix
- What happens if you maintain inclusion for L1I?

Level Two Cache Design

What is miss ratio?

- global: L2 misses after L1 references
- local: L2 misses after L1/L1 misses
- solo: as only cache / references

L1 cache design similar to single-level cache

- when main memory "faster" w.r.t. CPU

Apply previous experience to L2 design?

L2 "global" miss ratios not significantly altered by L1 presence

- if L2 cache size >= 8 x L1 cache size
- Przybylski et al., ISCA 1989

But L2 caches bigger than before

Level Two Cache Example

Recall adding associativity to single-level cache helped if

$diff(t_{cache}) + diff(\text{miss}) \times t_{miss} < 0$

$diff(\text{miss}) = -1\%$, $t_{miss} = 20$

$\Rightarrow diff(t_{cache}) < 0.2$ cycle

Consider doing the same in an L2 cache where

$t_{avg} = t_{cache1} + \text{miss1} \times t_{cache2} + \text{miss2} \times t_{memory}$

Improvement only if

$\text{miss1} \times diff(t_{cache2}) + diff(\text{miss2}) \times t_{memory} < 0$

$diff(t_{cache2}) < (-diff(\text{miss2})/\text{miss1}) \times t_{memory}$

$diff(t_{cache2}) < 0.0005/0.05 \times 100 = 1$ cycle