Reducing Conflict Misses with Set-Associative Caches

Not too conflict-y. Not too slow.

... Just Right!

8 byte, 2-way set associative Cache

XXXX

CACHE

index	V	tag	data	V	tag	data
0	0	xx	E | F	0	xx	N | O
1	0	xx	C | D	0	xx	P | Q

What should the **offset** be?

What should the **index** be?

What should the **tag** be?

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

8 byte, 2-way set associative Cache

XXXX

tag | index | offset

CACHE

index	V	tag	data	V	tag	data
0	0	xx	X | X	0	xx	X | X
1	0	xx	X | X	0	xx	X | X

```
load 0x1100   Miss
load 0x1101
load 0x0100
load 0x1100
```

Lookup:
- Index into $
- Check tag
- Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

8 byte, 2-way set associative Cache

XXXX

tag | index | offset

CACHE

index	V	tag	data	V	tag	data
0	1	11	N | O	0	xx	X | X
1	0	xx	X | X	0	xx	X | X

```
load 0x1100   Miss
load 0x1101   Hit!
load 0x0100
load 0x1100
```

Lookup:
- Index into $
- Check tag
- Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

8 byte, 2-way set associative Cache

XXXX

tag | index | offset

CACHE

index	V	tag	data	V	tag	data
0	1	11	N | O	0	xx	X | X
1	0	xx	X | X	0	xx	X | X

```
load 0x1100   Miss
load 0x1101   Hit!
load 0x0100   Miss
load 0x1100
```

Lookup:
- Index into $
- Check tag
- Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

8 byte, 2-way set associative Cache

XXXX

tag | index | offset

CACHE

index	V	tag	data	V	tag	data
0	1	11	N | O	1	01	E | F
1	0	xx	X | X	0	xx	X | X

```
load 0x1100   Miss
load 0x1101   Hit!
load 0x0100   Miss
load 0x1100   Hit!
```

Lookup:
- Index into $
- Check tag
- Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Misses: the Three C's

Cold (compulsory) Miss:
never seen this address before

Conflict Miss:
cache associativity is too low

Capacity Miss:
cache is too small

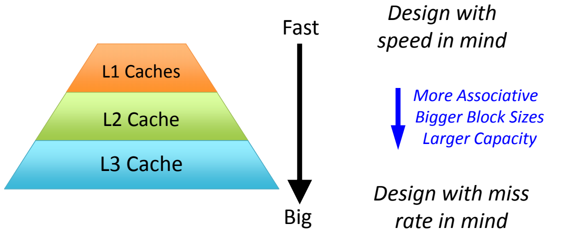
ABCs of Caches

$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$

+ Associativity:
  - ↓conflict misses ☺
  - ↑hit time ☹
+ Block Size:
  - ↓cold misses ☺
  - ↑conflict misses ☹
+ Capacity:
  - ↓capacity misses ☺
  - ↑hit time ☹

Which caches get what properties?

$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$

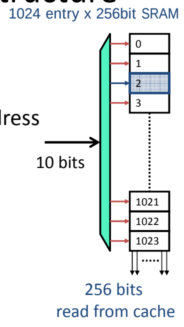


Summary so far

- Things we've covered:
  - The Need for Speed
  - Locality to the Rescue!
  - Calculating average memory access time
  - $ Misses: Cold, Conflict, Capacity
  - $ Characteristics: Associativity, Block Size, Capacity
- Things we skipped (and are about to cover):
  - Cache Overhead
  - Replacement Policies
  - Writes

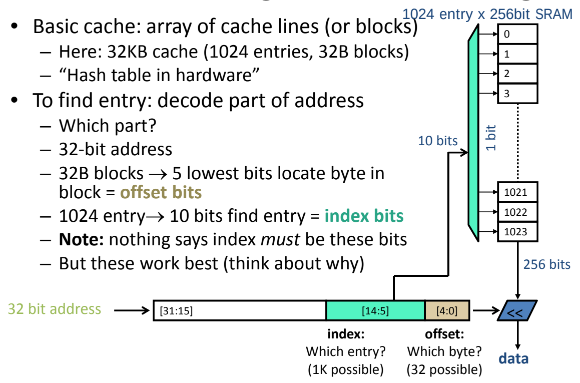
Basic Memory Array Structure

- Number of entries
  - n bits for lookup → 2^n entries
  - Example: 1024 entries, 10 bit address
  - Decoder changes n-bit address to 2^n bit "one-hot" signal
- Size of entries
  - Width of data accessed
  - Here: 256 bits (32 bytes)



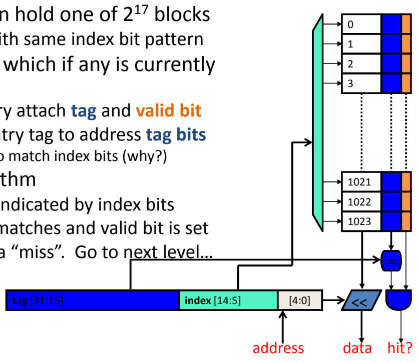
Caches: Finding Data via Indexing

- Basic cache: array of cache lines (or blocks)
  - Here: 32KB cache (1024 entries, 32B blocks)
  - "Hash table in hardware"
- To find entry: decode part of address
  - Which part?
  - 32-bit address
  - 32B blocks → 5 lowest bits locate byte in block = **offset bits**
  - 1024 entry→ 10 bits find entry = **index bits**
  - **Note:** nothing says index *must* be these bits
  - But these work best (think about why)



Knowing that You Found It: Tags

- Each entry can hold one of 2^{17} blocks
  - All blocks with same index bit pattern
- How to know which if any is currently there?
  - To each entry attach **tag** and **valid bit**
  - Compare entry tag to address **tag bits**
    - No need to match index bits (why?)
- Lookup algorithm
  - Read entry indicated by index bits
  - "Hit" if tag matches and valid bit is set
  - Otherwise, a "miss". Go to next level…



Calculating Tag Overhead

- "32KB cache" means cache holds 32KB of **data**
  - Called **capacity**
  - Tag storage is considered overhead
- Tag overhead of 32KB cache with 1024 x 32B entries
  - 32B blocks → ??-bit offset
  - 1024 entries → ??-bit index
  - 32-bit address → ??-bit tag
- What about 64-bit addresses?

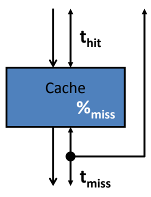
Calculating Tag Overhead

- "32KB cache" means cache holds 32KB of **data**
  - Called **capacity**
  - Tag storage is considered overhead
- Tag overhead of 32KB cache with 1024 x 32B entries
  - 32B blocks → 5-bit offset
  - 1024 entries → 10-bit index
  - 32-bit address → 32-bits – (5-bit offset + 10-bit index) = 17-bit tag
    (17-bit tag + 1-bit valid) X 1024 entries = 18Kb tags = 2.2KB tags
    ~6% overhead
- What about 64-bit addresses?
  - Tag increases to 49 bits, ~20% overhead

Handling a Cache Miss

- What if requested data isn't in the cache?
  - How does it get in there?
- **Cache controller**: finite state machine
  - Remembers miss address
  - Accesses next level of memory
  - Waits for response
  - Writes data/tag into proper locations
  - All of this happens on the **fill path**
  - Sometimes called **backside**

Cache Performance Equation



- **Access**: read or write to cache
- **Hit**: desired data found in cache
- **Miss**: desired data not found in cache
  - Must get from another component
  - No notion of "miss" in register file
- **Fill**: action of placing data into cache
- $\%_{miss}$ (miss-rate): #misses / #accesses
- t_{hit}: time to read data from (write data to) cache
- t_{miss}: time to read data into cache
- Performance metric: average access time

$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$

CPI Calculation with Cache Misses

- **Parameters**
  - Simple pipeline with base CPI of 1
  - Instruction mix: 30% loads/stores
  - I\$: $\%_{miss}$ = 2%, t_{miss} = 10 cycles
  - D\$: $\%_{miss}$ = 10%, t_{miss} = 10 cycles
- **What is new CPI?**
  - $CPI_{I\$}$ =
  - $CPI_{D\$}$ =
  - CPI_{new} =

CPI Calculation with Cache Misses

- **Parameters**
  - Simple pipeline with base CPI of 1
  - Instruction mix: 30% loads/stores
  - I\$: $\%_{miss}$ = 2%, t_{miss} = 10 cycles
  - D\$: $\%_{miss}$ = 10%, t_{miss} = 10 cycles
- **What is new CPI?**
  - $CPI_{I\$} = \%_{missI\$} * t_{miss}$ = 0.02*10 cycles = 0.2 cycle
  - $CPI_{D\$} = \%_{load/store} * \%_{missD\$} * t_{missD\$}$ = 0.3 * 0.1*10 cycles = 0.3 cycle
  - $CPI_{new} = CPI + CPI_{I\$} + CPI_{D\$}$ = 1+0.2+0.3 = 1.5

Measuring Cache Performance

- Ultimate metric is t_{avg}
  - Cache capacity and circuits roughly determines t_{hit}
  - Lower-level memory structures determine t_{miss}
  - Measure $\%_{miss}$
    - Hardware performance counters
    - Simulation
    - Paper simulation (next)

Cache Miss Paper Simulation

- 4-bit addresses → total memory size = 16B
  - Simpler cache diagrams than 32-bits
- 8B cache, 2B blocks
  - Number of entries (or *sets*): 4 (capacity / block-size)
  - Figure out how address splits into offset/index/tag bits
    - **Offset**: least-significant $\log_2$(block-size) = $\log_2(2)$ = 1 → `0000`
    - **Index**: next $\log_2$(number-of-entries) = $\log_2(4)$ = 2 → `0000`
    - **Tag**: rest = 4 – 1 – 2 = 1 → `0000`
- Cache diagram
  - `0000|0001` = addresses of data in block, values don't matter

Cache contents				Address	Outcome
Set00	Set01	Set10	Set11		
0000|0001	0010|0011	0100|0101	0110|0111		

Cache Miss Paper Simulation

8B cache, 2B blocks

tag (1 bit)	index (2 bits)	1 bit

Cache contents (prior to access)				Address	Outcome
Set00	Set01	Set10	Set11		
0000|0001	0010|0011	0100|0101	0110|0111	**1100**	Miss
				1110	
				1000	
				0011	
				1000	
				0000	
				1000	

- How to reduce $\%_{miss}$? And hopefully t_{avg}?

Cache Miss Paper Simulation

8B cache, 2B blocks

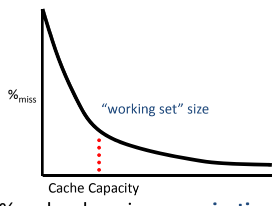
tag (1 bit)	index (2 bits)	1 bit

Cache contents (prior to access)				Address	Outcome
Set00	Set01	Set10	Set11		
0000|0001	0010|0011	0100|0101	0110|0111	**1100**	Miss
0000|0001	0010|0011	**1100|1101**	0110|0111	**1110**	Miss
0000|0001	0010|0011	1100|1101	**1110|1111**	**1000**	Miss
1000|1001	0010|0011	1100|1101	1110|1111	**0011**	Hit
1000|1001	0010|0011	1100|1101	1110|1111	**1000**	Hit
1000|1001	0010|0011	1100|1101	1110|1111	**0000**	Miss
0000|0001	0010|0011	1100|1101	1110|1111	**1000**	Miss

- How to reduce $\%_{miss}$? And hopefully t_{avg}?

Capacity and Performance

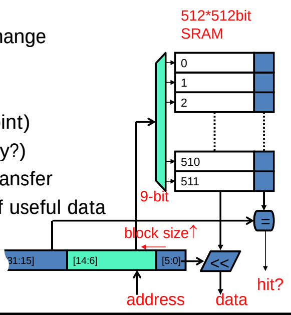
- Simplest way to reduce $\%_{miss}$: increase capacity
  + Miss rate decreases monotonically
    - **"Working set"**: insns/data program is actively using
    - Diminishing returns
  - However t_{hit} increases
    - Latency proportional to sqrt(capacity)
  - t_{avg} ?



- Given capacity, manipulate $\%_{miss}$ by changing **organization**

Block Size

- Given capacity, manipulate $\%_{miss}$ by changing organization
- One option: increase **block size**
  - Exploit **spatial locality**
  - Notice index/offset bits change
  - Tag remain the same
- Ramifications
  - + Reduce $\%_{miss}$ (up to a point)
  - + Reduce tag overhead (why?)
  - – Potentially useless data transfer
  - – Premature replacement of useful data
  - – Fragmentation



Block Size and Tag Overhead

- Tag overhead of 32KB cache with 1024 32B entries
  - 32B lines → 5-bit offset
  - 1024 entries → 10-bit index
  - 32-bit address →
- Tag overhead of 32KB cache with 512 64B entries
  - 64B lines →
  - 512 entries →
  - 32-bit address →

Block Size and Tag Overhead

- Tag overhead of 32KB cache with 1024 32B entries
  - 32B lines → 5-bit offset
  - 1024 entries → 10-bit index
  - 32-bit address → 32 – (5-bit offset + 10-bit index) = 17-bit tag
  - (17-bit tag + 1-bit valid) X 1024 entries = 18Kb tags = 2.2KB tags
  - ~6% overhead
- Tag overhead of 32KB cache with 512 64B entries
  - 64B lines → 6-bit offset
  - 512 entries → 9-bit index
  - 32-bit address → 32 – (6-bit offset + 9-bit index) = 17-bit tag
  - (17-bit tag + 1-bit valid) X 512 entries = 9Kb tags = 1.1KB tags
  - ~3% overhead

Block Size Cache Miss Paper Simulation

8B cache, **4B blocks**

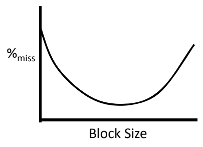
tag (1 bit)	**index** (1 bit1)	offset (2 bits)

Cache contents (prior to access)		Address	Outcome
Set0	Set1		
0000|0001|0010|0011	0100|0101|0110|0111	**1100**	Miss
0000|0001|0010|0011	**1100|1101|1110|1111**	**1110**	**Hit (spatial locality)**
0000|0001|0010|0011	1100|1101|1110|1111	**1000**	Miss
1000|1001|1010|1011	1100|1101|1110|1111	**0011**	**Miss**
0000|0001|0010|0011	1100|1101|1110|1111	**1000**	**Miss**
1000|1001|1010|1011	1100|1101|1110|1111	**0000**	Miss
0000|0001|0010|0011	1100|1101|1110|1111	**1000**	Miss

+ **Spatial "prefetching"**: miss on `1100` brought in `1110`
– **Conflicts**: miss on `1000` kicked out `0011`

Effect of Block Size on Miss Rate

- Two effects on miss rate
  - + **Spatial prefetching (good)**
    - For blocks with adjacent addresses
    - Turns miss/miss into miss/hit pairs
  - – **Interference (bad)**
    - For blocks with non-adjacent addresses (but in adjacent entries)
    - Turns hits into misses by disallowing simultaneous residence
    - Consider entire cache as one big block
- Both effects always present
  - Spatial prefetching dominates initially
    - Depends on size of the cache
  - Good block size is 16–128B
    - Program dependent



Block Size and Miss Penalty

- Does increasing block size increase t_{miss}?
  - Don't larger blocks take longer to read, transfer, and fill?
  - They do, but…
- t_{miss} of an isolated miss is not affected
  - **Critical Word First / Early Restart (CRF/ER)**
  - Requested word fetched first, pipeline restarts immediately
  - Remaining words in block transferred/filled in the background
- t_{miss}'es of a cluster of misses will suffer
  - Reads/transfers/fills of two misses can't happen at the same time
  - Latencies can start to pile up
  - This is a bandwidth problem (more later)

Conflicts

8B cache, 2B blocks

tag (1 bit)	index (2 bits)	1 bit

Cache contents (prior to access)				Address	Outcome
Set00	Set01	Set10	Set11		
0000|0001	0010|0011	0100|0101	0110|0111	**1100**	Miss
0000|0001	0010|0011	**1100|1101**	0110|0111	**1110**	Miss
0000|0001	0010|0011	1100|1101	**1110|1111**	**1000**	Miss
1000|1001	0010|0011	1100|1101	1110|1111	**0011**	Hit
1000|1001	0010|0011	1100|1101	1110|1111	**1000**	Hit
1000|1001	0010|0011	1100|1101	1110|1111	**0000**	Miss
0000|0001	0010|0011	1100|1101	1110|1111	**1000**	Miss

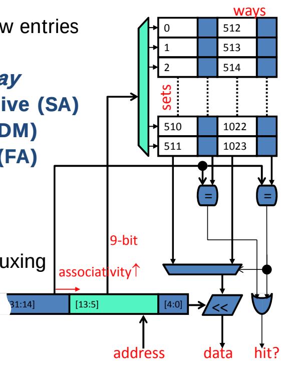
- Pairs like `0000/1000` **conflict**
  - Regardless of block-size (assuming capacity < 16)
  - Q: can we allow pairs like these to simultaneously reside?
  - A: yes, reorganize cache to do so

Set-Associativity

- Block can reside in one of few entries
- Entry groups called ***sets***
- Each entry in set called a ***way***
- This is **2-way set-associative (SA)**
- 1-way → **direct-mapped (DM)**
- 1-set → **fully-associative (FA)**

+ Reduces conflicts
– Increases latency$_{hit:}$
  - additional tag match & muxing

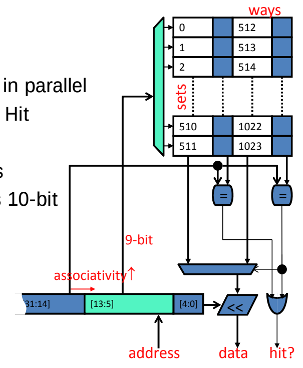
- Note: valid bit not shown



Set-Associativity

Lookup algorithm
- Use index bits to find set
- Read data/tags in all ways in parallel
- **Any** (match and valid bit), Hit

- Notice tag/index/offset bits
  - Only 9-bit index (versus 10-bit for direct mapped)
- Notice block numbering



Associativity and Miss Paper Simulation

8B cache, 2B blocks, **2-way set-associative**

Cache contents (prior to access)				Address	Outcome
Set0.Way0	Set0.Way1	Set1.Way0	Set1.Way1		
0000|0001	0100|0101	0010|0011	0110|0111	**1100**	Miss
				1110	
				1000	
				0011	
				1000	
				0000	
				1000	

+ **Avoid conflicts**: `0000` and `1000` can both be in set 0
– **Introduce some new conflicts**: addresses get re-arranged
  - Conflict avoidance usually dominates

Associativity and Miss Paper Simulation

8B cache, 2B blocks, **2-way set-associative**

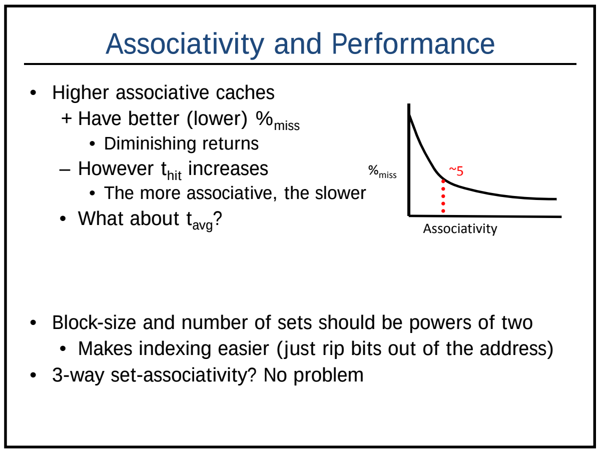
Cache contents (prior to access)				Address	Outcome
Set0.Way0	Set0.Way1	Set1.Way0	Set1.Way1		
0000|0001	0100|0101	0010|0011	0110|0111	**1100**	Miss
1100|1101	0100|0101	0010|0011	0110|0111	**1110**	Miss
1100|1101	0100|0101	**1110|1111**	0110|0111	**1000**	Miss
1100|1101	**1000|1001**	1110|1111	0110|0111	**0011**	**Miss (new conflict)**
1100|1101	1000|1001	1110|1111	**0010|0011**	**1000**	Hit
1100|1101	1000|1001	1110|1111	0010|0011	**0000**	Miss
0000|0001	1000|1001	1110|1111	0010|0011	**1000**	**Hit (avoid conflict)**

+ **Avoid conflicts**: `0000` and `1000` can both be in set 0
– **Introduce some new conflicts**: addresses get re-arranged
  - Conflict avoidance usually dominates

Replacement Policies

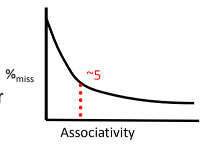
- Associative caches present a new design choice
  - On cache miss, which block in set to replace (kick out)?
- Some options
  - **Random**
  - **FIFO (first-in first-out)**
  - **LRU (least recently used)**
    - Fits with temporal locality, LRU = least likely to be used in future
  - **NMRU (not most recently used)**
    - An easier to implement approximation of LRU
    - Is LRU for 2-way set-associative caches
  - **Belady's**: replace block that will be used furthest in future
    - Unachievable optimum

  - Which policy is simulated in previous example?



Associativity and Performance

- Higher associative caches
  - + Have better (lower) $\%_{miss}$
    - Diminishing returns
  - – However t_{hit} increases
    - The more associative, the slower
  - What about t_{avg}?



- Block-size and number of sets should be powers of two
  - Makes indexing easier (just rip bits out of the address)
- 3-way set-associativity? No problem