CSE 5600
Computer Systems Architecture

Cache

1

Why Caches?

2

Programs 101

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated IA32 Assembly

```
sum:
   pushl %ebp
   movl %esp,%ebp
   movl 12(%ebp),%eax
   addl 8(%ebp),%eax
   popl %ebp
   ret
```

High-level behavior: Instructions that read from/write to memory…

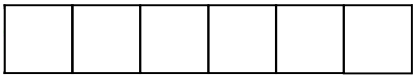
- Read data from memory (put in registers)
- Manipulate it
- Store it back to memory

3

3

The Need for Speed

CPU Pipeline

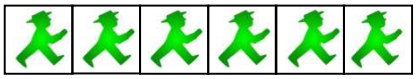


4

4

The Need for Speed

CPU Pipeline



Instruction speeds:

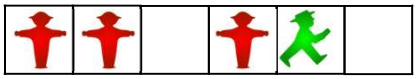
- `add,sub,shift`: 1 cycle
- `mult`: 3 cycles
- `load/store`: **100 cycles**
  - off-chip 50(-70)ns
  - 2(-3) GHz processor → 0.5 ns clock

5

5

The Need for Speed

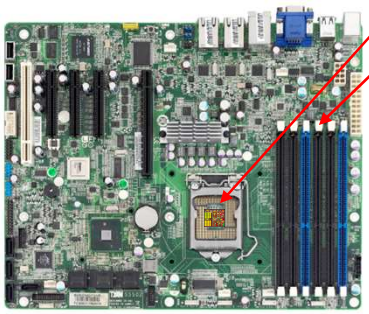
CPU Pipeline



6

6

What's the problem?



Processor

Main Memory
- too slow
- too far away

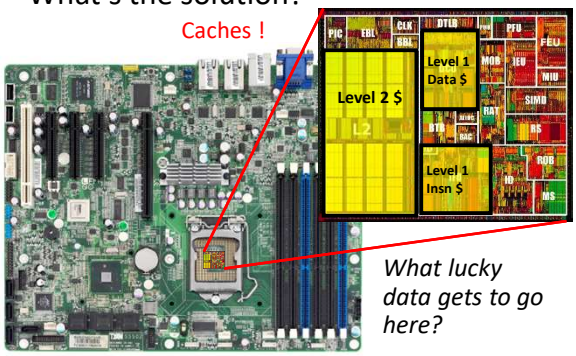
SandyBridge Motherboard, 2011
http://news.softpedia.com

7

7

What's the solution?

Caches !



What lucky data gets to go here?

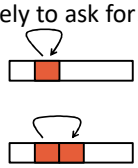
Intel Pentium 3, 1999

8

8

Locality Locality Locality

If you ask for something, you're likely to ask for:
- the same thing again soon
  → Temporal Locality
- something near that thing, soon
  → Spatial Locality



```
total = 0;
for (i = 0; i < n; i++)
    total += a[i];
return total;
```

9

9

Your life is full of Locality

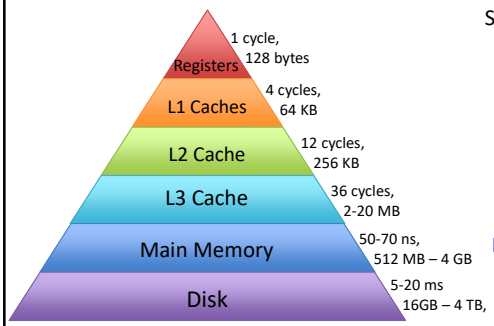




10

10

The Memory Hierarchy



Small, Fast

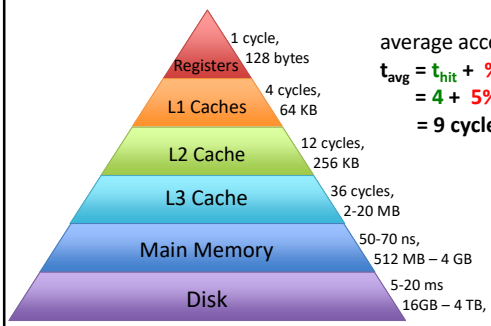
Big, Slow

Intel Haswell Processor, 2013

11

11

The Memory Hierarchy



average access time

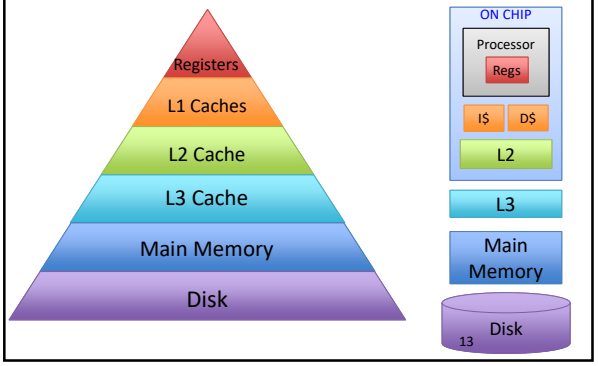
$t_{avg} = t_{hit} + \%_{miss} \times t_{miss}$
$= 4 + 5\% \times 100$
$= 9$ cycles

Intel Haswell Processor, 2013

12

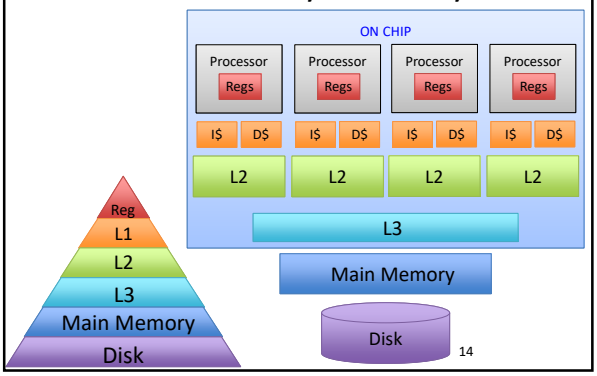
12

The Memory Hierarchy



13

The Memory Hierarchy



14

Basic Cache Design

Direct Mapped Caches



15

16 Byte Memory

```
load 0x1100 → r1
```

- Byte-addressable memory
- 4 address bits → 16 bytes total
- b addr bits → 2^b bytes in memory

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

16

4-Byte, Direct Mapped Cache

CACHE

index	addr	data
00	xxxx	X
01	xxxx	X
10	xxxx	X
11	xxxx	X

- entry = row = **cache line** = **cache block**
- **Block Size:** 1 byte
- **Direct mapped:**
  - Each address mapped to specific cache block
  - 4 entries → 2 index bits (2^n → n bits)

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

17

Least Significant Bits as Index

index
XXXX

CACHE

index	addr	data
00	0000	A
01	0001	B
10	0010	C
11	0011	D

- Supports spatial locality

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

18

4-Byte, Direct Mapped Cache

tag | index
XXXX

CACHE

index	tag	data
00	00	A
01	00	B
10	00	C
11	00	D

Tag: minimalist label/address

`address = tag + index`

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

21

Simulation #1 of a 4-byte, DM Cache

tag | index
XXXX

CACHE

index	tag	data
00	00	0
01	00	0
10	00	0
11	00	0

Cache starts empty

`load 0x0000` Hit?

Lookup:
- Index into $
- Check tag

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

22

4-Byte, Direct Mapped Cache

CACHE

index	V	tag	data
00	0	xx	X
01	0	xx	X
10	0	xx	X
11	0	xx	X

One last tweak: **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

23

Simulation #1 of a 4-byte, DM Cache

tag | index
XXXX

CACHE

index	V	tag	data
00	0	xx	X
01	0	xx	X
10	0	xx	X
11	0	xx	X

`load 0x1100` Miss

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

24

Simulation #1 of a 4-byte, DM Cache

tag | index
XXXX

CACHE

index	V	tag	data
00	1	11	N
01	0	xx	X
10	0	xx	X
11	0	xx	X

`load 0x1100` Miss

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

25

Simulation #1 of a 4-byte, DM Cache

tag | index
XXXX

CACHE

index	V	tag	data
00	1	11	N
01	0	xx	X
10	0	xx	X
11	0	xx	X

`load 0x1100` Miss

`...`

`load 0x1100` Hit!

Awesome!

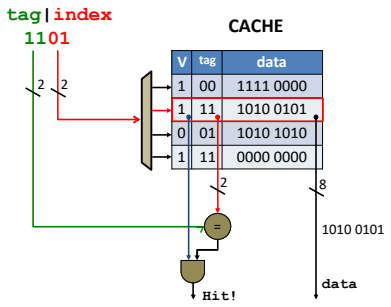
Lookup:
- Index into $
- Check tag
- Check valid bit

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

26

Block Diagram
4-entry, direct mapped Cache

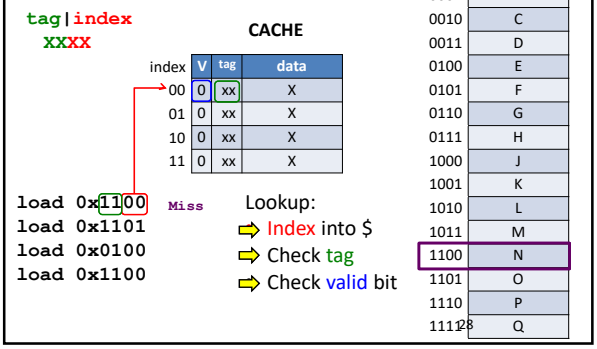


Great!
Are we done?

27

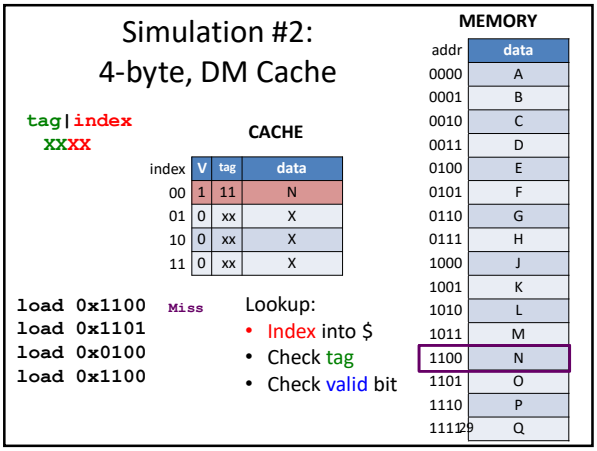
27

Simulation #2:
4-byte, DM Cache



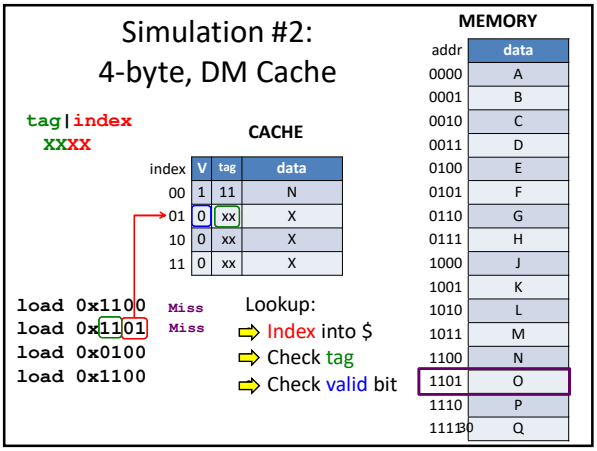
28

Simulation #2:
4-byte, DM Cache



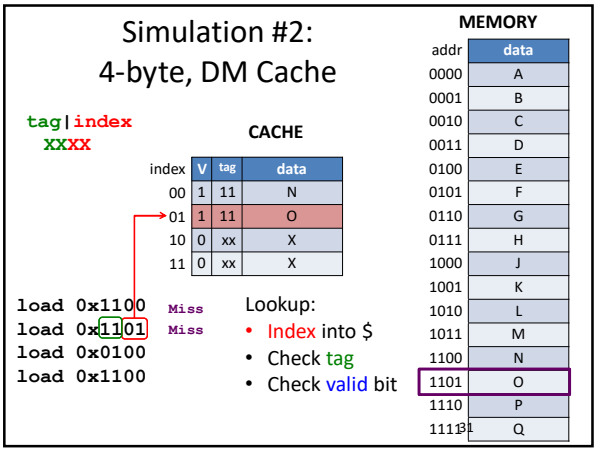
29

Simulation #2:
4-byte, DM Cache



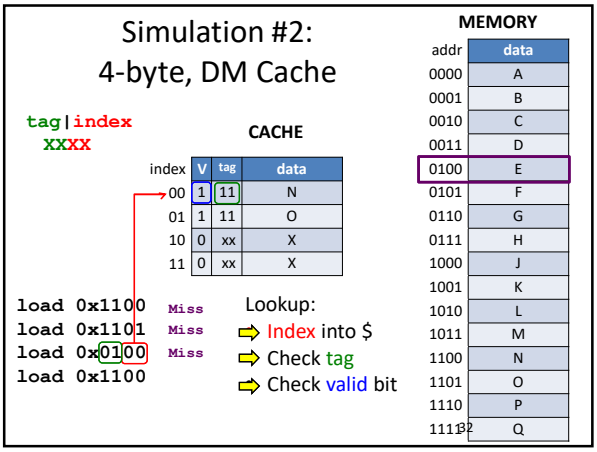
30

Simulation #2:
4-byte, DM Cache



31

Simulation #2:
4-byte, DM Cache



32

Simulation #2: 4-byte, DM Cache

tag | index
XXXX

CACHE

index	V	tag	data
00	1	01	E
01	1	11	O
10	0	xx	X
11	0	xx	X

```
load 0x1100   Miss
load 0x1101   Miss
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

Simulation #2: 4-byte, DM Cache

tag | index
XXXX

CACHE

index	V	tag	data
00	1	01	E
01	1	11	O
10	0	xx	X
11	0	xx	X

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

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

Simulation #2: 4-byte, DM Cache

tag | index
XXXX

CACHE

index	V	tag	data
00	1	11	N
01	1	11	O
10	0	xx	X
11	0	xx	X

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

Disappointed!

☹

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

Reducing Cold Misses by Increasing Block Size

Leveraging Spatial Locality

Increasing Block Size

CACHE

offset
XXXX

index	V	tag	data
00	0	x	A | B
01	0	x	C | D
10	0	x	E | F
11	0	x	G | H

- **Block Size:** 2 bytes
- **Block Offset:** least significant bits indicate where you live in the block
- Which bits are the index? tag?

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

Simulation #3: 8-byte, DM Cache

tag | index | offset
XXXX

CACHE

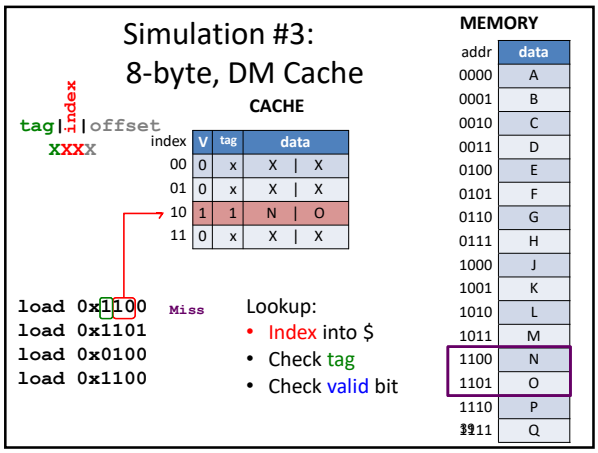
index	V	tag	data
00	0	x	X | X
01	0	x	X | X
10	0	x	X | X
11	0	x	X | X

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

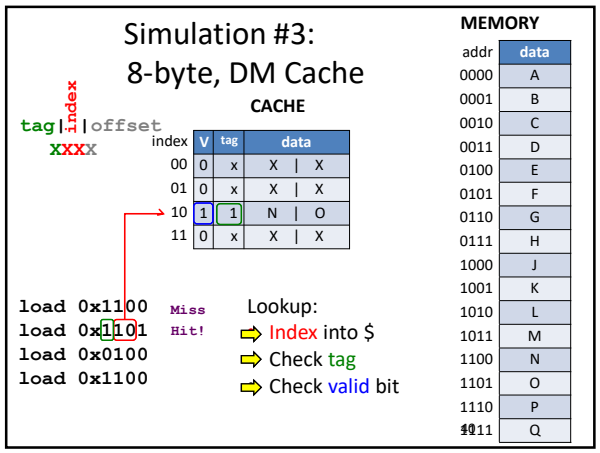
Lookup:
- ⇨ Index into $
- ⇨ Check tag
- ⇨ Check valid bit

MEMORY

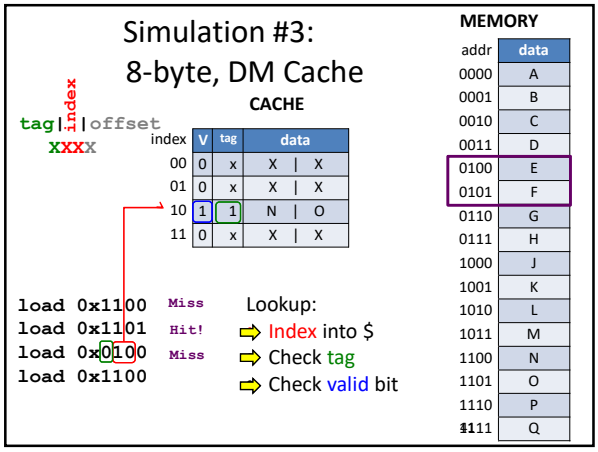
addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q



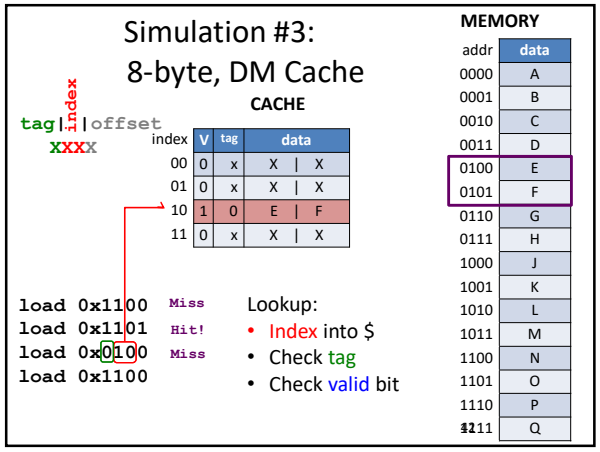
39



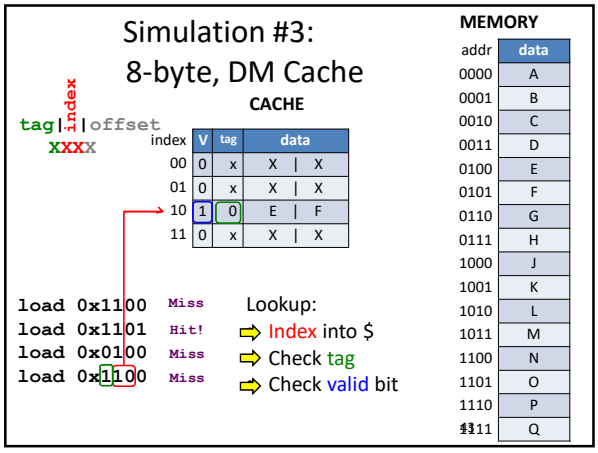
40



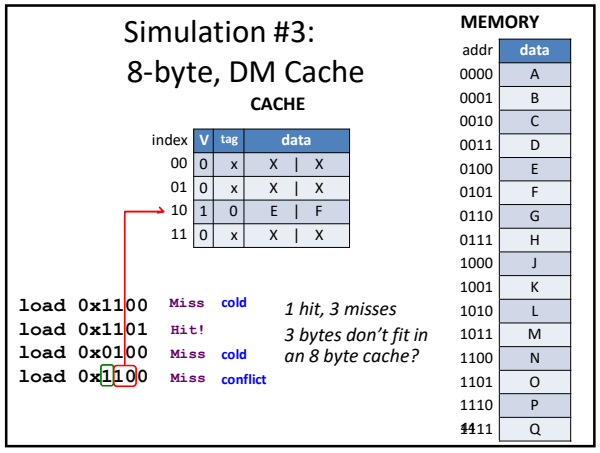
41



42



43



44

Removing Conflict Misses with Fully-Associative Caches



45

8 byte, fully-associative Cache

XXXX

CACHE

V	tag	data	V	tag	data	V	tag	data	V	tag	data
0	xxx	X | X	0	xxx	X | X	0	xxx	X | X	0	xxx	X | X

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

46

Simulation #4: 8-byte, FA Cache

XXXX

tag|offset

CACHE

V	tag	data	V	tag	data	V	tag	data	V	tag	data
0	xxx	X | X	0	xxx	X | X	0	xxx	X | X	0	xxx	X | X

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

Lookup:
- ~~Index into $~~
- ⇨ Check tags
- ⇨ Check valid bits

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

47

Simulation #4: 8-byte, FA Cache

XXXX

tag|offset

CACHE

V	tag	data	V	tag	data	V	tag	data	V	tag	data
1	110	N | O	0	xxx	X | X	0	xxx	X | X	0	xxx	X | X

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

Lookup:
- ~~Index into $~~
- ⇨ Check tags
- ⇨ Check valid bits

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

48

Simulation #4: 8-byte, FA Cache

XXXX

tag|offset

CACHE

V	tag	data	V	tag	data	V	tag	data	V	tag	data
1	110	N | O	0	xxx	X | X	0	xxx	X | X	0	xxx	X | X

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

Lookup:
- ~~Index into $~~
- ⇨ Check tags
- ⇨ Check valid bits

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

49

Simulation #4: 8-byte, FA Cache

XXXX

tag|offset

CACHE

V	tag	data	V	tag	data	V	tag	data	V	tag	data
1	110	N | O	1	010	E | F	0	xxx	X | X	0	xxx	X | X

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

Lookup:
- ~~Index into $~~
- ⇨ Check tags
- ⇨ Check valid bits

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

50

Pros and Cons of Full Associativity

+ No more conflicts!
+ Excellent utilization!

But…

Parallel Reads
- lots of reading!

Serial Reads
- lots of waiting



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

= 4 + 5% x 100 = 9 cycles

= 6 + 3% x 100 = 9 cycles

51

51

Reducing Conflict Misses with Set-Associative Caches

Not too conflict-y. Not too slow.

… Just Right!



52

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

53

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

54

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

55

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

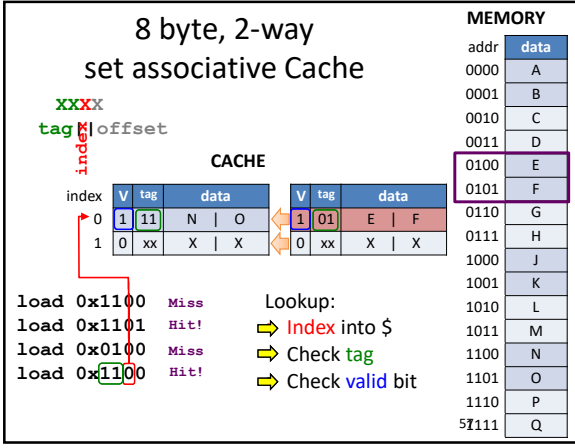
Lookup:
- Index into $
- Check tag
- Check valid bit

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

56

8 byte, 2-way set associative Cache



57

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

58

58

ABCs of Caches



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

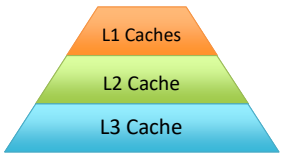
+ Associativity:
  - ↓conflict misses ☺
  - ↑hit time ☹
+ Block Size:
  - ↓cold misses ☺
  - ↑conflict misses ☹
+ Capacity:
  - ↓capacity misses ☺
  - ↑hit time ☹

59

59

Which caches get what properties?

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



Fast

Big

Design with speed in mind

More Associative
Bigger Block Sizes
Larger Capacity

Design with miss rate in mind

60

60

Summary so far

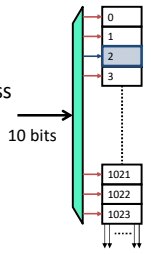
- Things we've covered:
  - The Need for Speed
  - Locality to the Rescue!
  - Calculating average memory access time
  - $ Misses: Cold, Conflict, Capacity
  - $ Characteristics: Associativity, Block Size, Capacity
- Things we skipped (and are about to cover):
  - Cache Overhead
  - Replacement Policies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

61

61

Basic Memory 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)

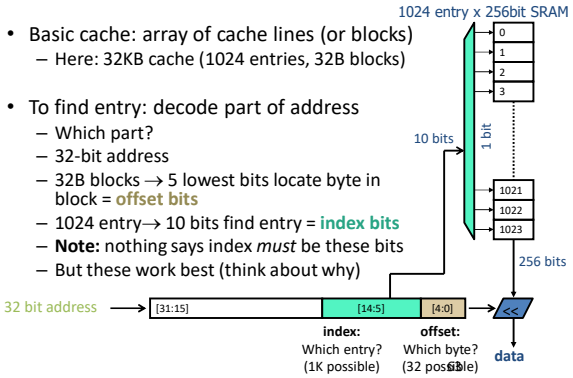


62

62

Caches: Finding Data via Indexing

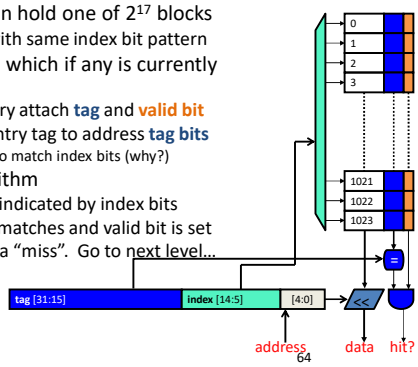
- Basic cache: array of cache lines (or blocks)
  - Here: 32KB cache (1024 entries, 32B blocks)
- To find entry: decode part of address
  - Which part?
  - 32-bit address
  - 32B blocks → 5 lowest bits locate byte in block = **offset bits**
  - 1024 entry→ 10 bits find entry = **index bits**
  - **Note:**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



63

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…



64

63

64

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?

65

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

66

65

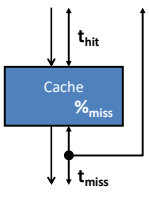
66

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**

67

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

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

68

67

68

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} =

69

69

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?**
  - $CPI_{I\$} = \%_{missI\$} \times t_{miss}$ = 0.02 x 10 cycles = 0.2 cycle
  - $CPI_{D\$} = \%_{load/store} \times \%_{missD\$} \times t_{missD\$}$ = 0.3 x 0.1 x 10 cycles = 0.3 cycle
  - $CPI_{new} = CPI + CPI_{I\$} + CPI_{D\$}$ = 1+0.2+0.3 = 1.5

70

70

Measuring Cache Performance

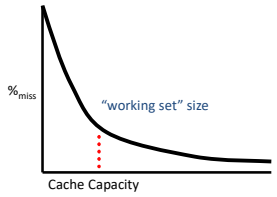
- Ultimate metric is t_{avg}
  - Cache capacity and circuits roughly determines t_{hit}
  - Lower-level memory structures determine t_{miss}
  - Measure $\%_{miss}$
    - Hardware performance counters
    - Simulation

71

71

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} ?



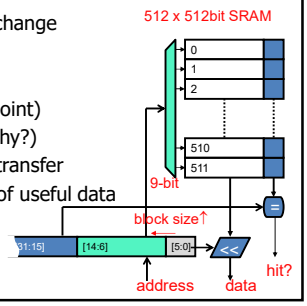
- Given capacity, manipulate $\%_{miss}$ by changing **organization**

75

75

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



76

76

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 →

77

77

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

78

Block Size and Miss Penalty

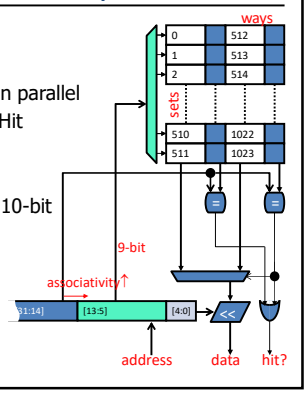
- Does increasing block size increase t_{miss}?
  - Don't larger blocks take longer to read, transfer, and fill?
  - They do, but…
- t_{miss} of an isolated miss is not affected
  - **Critical Word First / Early Restart (CRF/ER)**
  - Requested word fetched first, pipeline restarts immediately
  - Remaining 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)

81

Set-Associativity

Lookup algorithm

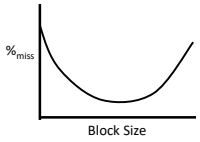
- Use index bits to find set
- Read data/tags in all ways in parallel
- **Any** (match and valid bit), Hit
- Notice tag/index/offset bits
  - Only 9-bit index (versus 10-bit for direct mapped)
- Notice block numbering



84

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

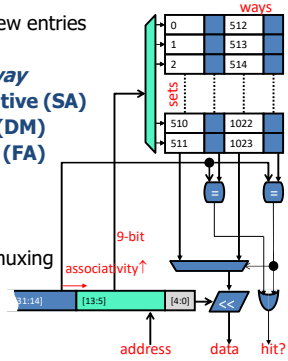


80

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



83

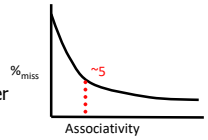
Replacement Policies

- 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

87

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

88

Classifying Misses: 3C Model (Hill)

- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - **Would miss even in infinite cache**
  - **Capacity**: miss caused because cache is too small
    - **Would miss even in fully associative cache**
    - Identify? Consecutive accesses to block separated by access to at least N other distinct blocks (N is number of entries in cache)
  - **Conflict**: miss caused because cache associativity is too low
    - Identify? **All other misses**
  - **(Coherence)**: miss due to external invalidations
    - Only in shared memory multiprocessors (later)
- Calculated by multiple simulations
  - Simulate infinite cache, fully-associative cache, normal cache
  - Subtract to find each count

89

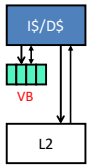
Miss Rate: ABC

- Why do we care about 3C miss model?
  - So that we know what to do to eliminate misses
  - If you don't have conflict misses, increasing associativity won't help
- **Associativity**
  - + Decreases conflict misses
  - – Increases $latency_{hit}$
- **Block size**
  - – Increases conflict/capacity misses (fewer entries)
  - + Decreases compulsory/capacity misses (spatial locality)
  - No significant effect on $latency_{hit}$
- **Capacity**
  - + Decreases capacity misses
  - – Increases $latency_{hit}$

90

Reducing Conflict Misses: Victim Buffer

- Conflict misses: not enough associativity
  - High-associativity is expensive, but also rarely needed
    - 3 blocks mapping to same 2-way set and accessed (XYZ)+
- **Victim buffer (VB)**: small fully-associative cache
  - Sits on I$/D$ miss path
  - Small so very fast (e.g., 8 entries)
  - Blocks kicked out of I$/D$ placed in VB
  - On miss, check VB: hit? Place block back in I$/D$
  - 8 extra ways, shared among all sets
    - + Only a few sets will need it at any given time
  - + Very effective in practice
  - Does VB reduce **$\%_{miss}$** or **$latency_{miss}$**?



91

Overlapping Misses: Lockup Free Cache

- **Lockup free**: allows other accesses while miss is pending
  - Consider: `load r2←[r1]; load r4←[r3]; add r5←r2,r4`
  - **Handle misses in parallel**
    - "memory-level parallelism"
  - Makes sense for…
    - Processors that can go ahead despite D$ miss (out-of-order)
  - Implementation: **miss status holding register (MSHR)**
    - Remember: miss address, chosen entry, requesting instruction
    - When miss returns know where to put block, who to inform
  - Common scenario: "hit under miss"
    - Handle hits while miss is pending
    - Easy
  - Less common, but common enough: "miss under miss"
    - A little trickier, but common anyway
    - Requires multiple MSHRs: search to avoid frame conflicts

92

Software Restructuring: Data

- Capacity misses: poor spatial or temporal locality
  - Several code restructuring techniques to improve both
  - – Compiler must know that restructuring preserves semantics

Loop interchange: spatial locality

- Example: row-major matrix: `X[i][j]` followed by `X[i][j+1]`
- Poor code: `X[i][j]` followed by `X[i+1][j]`

```
for (j = 0; j<NCOLS; j++)
   for (i = 0; i<NROWS; i++)
      sum += X[i][j];    // say
```

- Better code

```
for (i = 0; i<NROWS; i++)
   for (j = 0; j<NCOLS; j++)
      sum += X[i][j];
```

93

Software Restructuring: Data

- **Loop blocking**: temporal locality
  - Poor code

```
for (k=0; k<NITERATIONS; k++)
   for (i=0; i<NELEMS; i++)
      sum += X[i];    // say
```

  - Better code
    - Cut array into CACHE_SIZE chunks
    - Run all phases on one chunk, proceed to next chunk

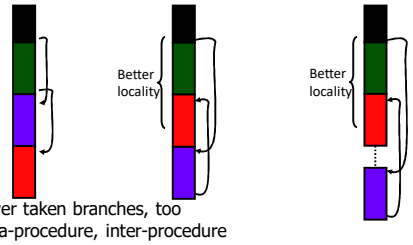
```
for (i=0; i<NELEMS; i+=CACHE_SIZE)
   for (k=0; k<NITERATIONS; k++)
      for (ii=0; ii<i+CACHE_SIZE-1; ii++)
         sum += X[ii];
```

  - Assumes you know `CACHE_SIZE`, do you?
  - Loop fusion: similar, but for multiple consecutive loops

94

Software Restructuring: Code

- Compiler can layout code for temporal and spatial locality
  - If (a) { **code1;** } else { **code2;** } **code3;**
  - But, code2 case never happens (say, error condition)



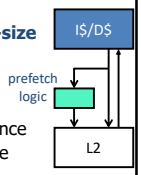
- Fewer taken branches, too
- Intra-procedure, inter-procedure

95

Prefetching

Prefetching: put blocks in cache proactively/speculatively

- Key: anticipate upcoming miss addresses accurately
  - Can do in software or hardware
- Simple example: **next block prefetching**
  - Miss on addr **X** → anticipate miss on **X+block-size**
  - \+ Works for insns: sequential execution
  - \+ Works for data: arrays



- **Timeliness**: initiate prefetches sufficiently in advance
- **Coverage**: prefetch for as many misses as possible
- **Accuracy**: don't pollute with unnecessary data
  - It evicts useful data

96

Software Prefetching

- Use a special "prefetch" instruction
  - Tells the hardware to bring in data, doesn't actually read it
  - Just a hint
- Inserted by programmer or compiler

Example:

```
for (i = 0; i<NROWS; i++)
   for (j = 0; j<NCOLS; j+=BLOCK_SIZE) {
      __builtin_prefetch(&X[i][j]+BLOCK_SIZE);
      for (jj=j; jj<j+BLOCK_SIZE-1; jj++)
         sum += x[i][jj];
   }
```

- Multiple prefetches bring multiple blocks in parallel
  - Using lockup-free caches
  - "Memory-level" parallelism

98

Hardware Prefetching

- What to prefetch?
  - **Stride-based sequential prefetching**
    - Can also do N blocks ahead to hide more latency
    - \+ Simple, works for sequential things: insns, array data
    - \+ Works better than doubling the block size
  - **Address-prediction**
    - Needed for non-sequential data: lists, trees, etc.
    - Use a hardware table to detect strides, common patterns
- When to prefetch?
  - On every reference?
  - On every miss?

99

More Advanced Address Prediction

- "Next-block" prefetching is easy, what about other options?
- **Correlating predictor**
  - Large table stores (miss-addr → next-miss-addr) pairs
  - On miss, access table to find out what will miss next
    - It's OK for this table to be large and slow
- Content-directed or dependence-based prefetching
  - Greedily chases pointers from fetched blocks
- Jump pointers
  - Augment data structure with prefetch pointers
- Make it easier to prefetch: cache-conscious layout/malloc
  - Lays lists out serially in memory, so they look like arrays
- Active area of research

100

Write Issues

- So far we have looked at reading from cache
  - Instruction fetches, loads
- What about writing into cache
  - Stores, not an issue for instruction caches (why they are simpler)
- Several new issues
  - Tag/data access
  - Write-through vs. write-back
  - Write-allocate vs. write-not-allocate
  - Hiding write miss latency

101

Tag/Data Access

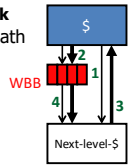
- Reads: read tag and data in parallel
  - Tag mis-match → data is garbage (OK, stall until good data arrives)
- Writes: read tag, write data in parallel?
  - Tag mis-match → clobbered data (oops)
  - For associative caches, which way was written into?
- Writes are a pipelined two step (multi-cycle) process
  - Step 1: match tag
  - Step 2: write to matching way
  - Bypass (with address check) to avoid load stalls
  - May introduce structural hazards

102

Write Propagation

When to propagate new value to (lower level) memory?

- **Option #1: Write-through**: immediately
  - On hit, update cache
  - Immediately send the write to the next level
- **Option #2: Write-back**: when block is replaced
  - Requires additional "dirty" bit per block
    - Replace **clean** block: **no extra traffic**
    - Replace **dirty** block: **extra "writeback" of block**
  - + **Writeback-buffer (WBB)**: keep it off critical path
    1. Send "fill" request to next-level
    2. While waiting, write dirty block to buffer
    3. When new blocks arrives, put it into cache
    4. Write buffer contents to next-level



103

Write Propagation Comparison

- **Write-through**
  - – Requires additional bus bandwidth
    - Consider repeated write hits
  - – Next level must handle small writes (1, 2, 4, 8-bytes)
  - + No need for dirty bits in cache
  - + No need to handle "writeback" operations
    - Simplifies miss handling (no write-back buffer)
  - Sometimes used for L1 caches (for example, by IBM)
- **Write-back**
  - + Key advantage: uses less bandwidth
  - Reverse of other pros/cons above
  - Used by Intel and AMD
  - 2nd-level and beyond are generally write-back caches

104

Write Miss Handling

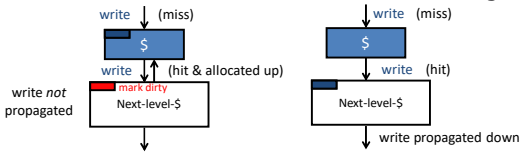
How is a write miss handled?

Write-allocate: fill block from next level, then write it

- + Decreases read misses (next read to block will hit)
- – Requires additional bandwidth
- Commonly used (especially with **write-back** caches)

Write-non-allocate: just write to next level, no allocate

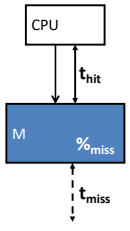
- – Potentially more read misses
- + Uses less bandwidth
- Use with **write-through**



105

105

Memory Performance Equation

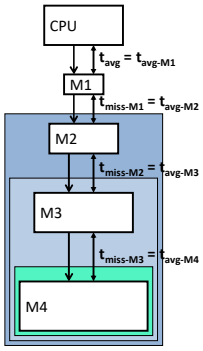


- **Access**: read or write to M
- **Hit**: desired data found in M
- **Miss**: desired data not found in M
  - Must get from another (slower) component
- **Fill**: action of placing data in M
- $\%_{miss}$ (miss-rate): #misses / #accesses
- t_{hit}: time to read data from (write data to) M
- t_{miss}: time to read data into M
- Performance metric
  - t_{avg}: average access time

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

106

Hierarchy Performance



t_{avg} =
t_{avg-M1}
$t_{hit-M1} + (\%_{miss-M1} \times t_{miss-M1})$
$t_{hit-M1} + (\%_{miss-M1} \times t_{avg-M2})$
$t_{hit-M1} + (\%_{miss-M1} \times (t_{hit-M2} + (\%_{miss-M2} \times t_{miss-M2})))$
$t_{hit-M1} + (\%_{miss-M1} \times (t_{hit-M2} + (\%_{miss-M2} \times t_{avg-M3})))$
…

107

Performance Calculation with $ Hierarchy

- **Parameters**
  - Reference stream: all loads
  - D$: t_{hit} = 1ns, $\%_{miss}$ = 5%
  - L2: t_{hit} = 10ns, $\%_{miss}$ = 20% (local miss rate)
  - Main memory: t_{hit} = 50ns
- **What is $t_{avgD\$}$ without an L2?**
  - $t_{missD\$}$ =
  - $t_{avgD\$}$ =
- **What is $t_{avgD\$}$ with an L2?**
  - $t_{missD\$}$ =
  - t_{avgL2} =
  - $t_{avgD\$}$ =

108

Performance Calculation with $ Hierarchy

- **Parameters**
  - Reference stream: all loads
  - D$: t_{hit} = 1ns, $\%_{miss}$ = 5%
  - L2: t_{hit} = 10ns, $\%_{miss}$ = 20% (local miss rate)
  - Main memory: t_{hit} = 50ns
- **What is $t_{avgD\$}$ without an L2?**
  - $t_{missD\$} = t_{hitM}$
  - $t_{avgD\$} = t_{hitD\$} + \%_{missD\$} \times t_{hitM}$ = 1ns+(0.05x50ns) = 3.5ns
- **What is $t_{avgD\$}$ with an L2?**
  - $t_{missD\$} = t_{avgL2}$
  - $t_{avgL2} = t_{hitL2} + \%_{missL2} \times t_{hitM}$ = 10ns+(0.2x50ns) = 20ns
  - $t_{avgD\$} = t_{hitD\$} + \%_{missD\$} \times t_{avgL2}$ = 1ns+(0.05x20ns) = 2ns

109

Designing a Cache Hierarchy

- For any memory component: t_{hit} vs. $\%_{miss}$ tradeoff
- Upper components (I$, D$) emphasize low t_{hit}
  - Frequent access → t_{hit} important
  - t_{miss} is not bad → $\%_{miss}$ less important
  - Low capacity/associativity (to reduce t_{hit})
  - Small-medium block-size (to reduce conflicts)
- Moving down (L2, L3) emphasis turns to $\%_{miss}$
  - Infrequent access → t_{hit} less important
  - t_{miss} is bad → $\%_{miss}$ important
  - High capacity/associativity/block size (to reduce $\%_{miss}$)

110

Split vs. Unified Caches

Split I$/D$: insns and data in different caches
- To minimize structural hazards and t_{hit}
- Larger unified I$/D$ would be slow, 2nd port even slower
- Optimize I$ for wide output (superscalar), no writes

Unified L2, L3: insns and data together
- To minimize $\%_{miss}$
+ Fewer capacity misses: unused insn capacity used for data
– More conflict misses: insn/data conflicts
  - A much smaller effect in large caches
- Insn/data structural hazards are rare: simultaneous I$/D$ miss
- Go even further: unify L2, L3 of multiple cores in a multi-core

112

Hierarchy: Inclusion versus Exclusion

- **Inclusion**
  - A block in the L1 is always in the L2
  - Good for write-through L1s (why?)
- **Exclusion**
  - Block is either in L1 or L2 (never both)
  - Good if L2 is small relative to L1
    - Example: AMD's Duron 64KB L1s, 64KB L2
- **Non-inclusion**
  - No guarantees

113

Summary

- **Average access time** of a memory component
  - $latency_{avg}$ = $latency_{hit}$ + $\%_{miss}$ x $latency_{miss}$
  - low $latency_{hit}$ and $\%_{miss}$ in one structure = hard → hierarchy
- **Memory hierarchy**
  - Cache (SRAM) → memory (DRAM) → swap (Disk)
  - Smaller, faster, more expensive → bigger, slower, cheaper
- Cache ABCs (**associativity, block size, capacity**)
  - 3C miss model: compulsory, capacity, conflict
- **Performance optimizations**
  - $\%_{miss}$: prefetching
  - $latency_{miss}$: victim buffer, critical-word-first, lockup-free design
- **Write issues**
  - Write-back vs. write-through
  - write-allocate vs. write-no-allocate

114