Direct-Mapped Cache

Big and Slow Meets Small and Fast

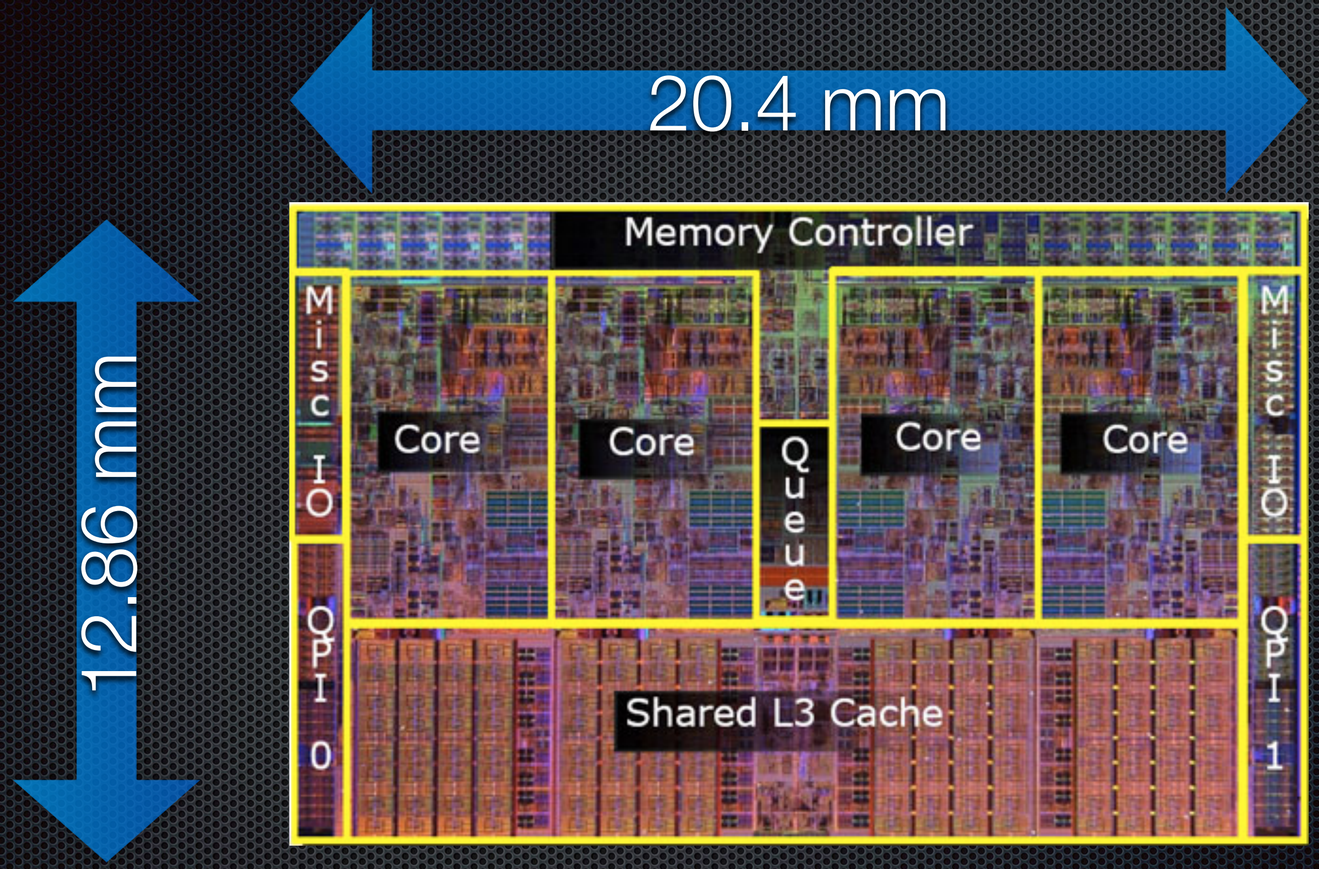
Chip Space Distribution

Intel Core i7

Core i7 Specs

- 4 cores, 2.66 to 3.2 GHz
- 731 million transistors, 45nm process
- Die size 263mm^2 = 20.40mm x 12.86mm = 0.8 x 0.5 in
- 32K L1 I-cache, 32K L1 D-cache, 1M L2, 8M L3
- On-chip memory controller w/3 channels, DDR3
- 1366 pins

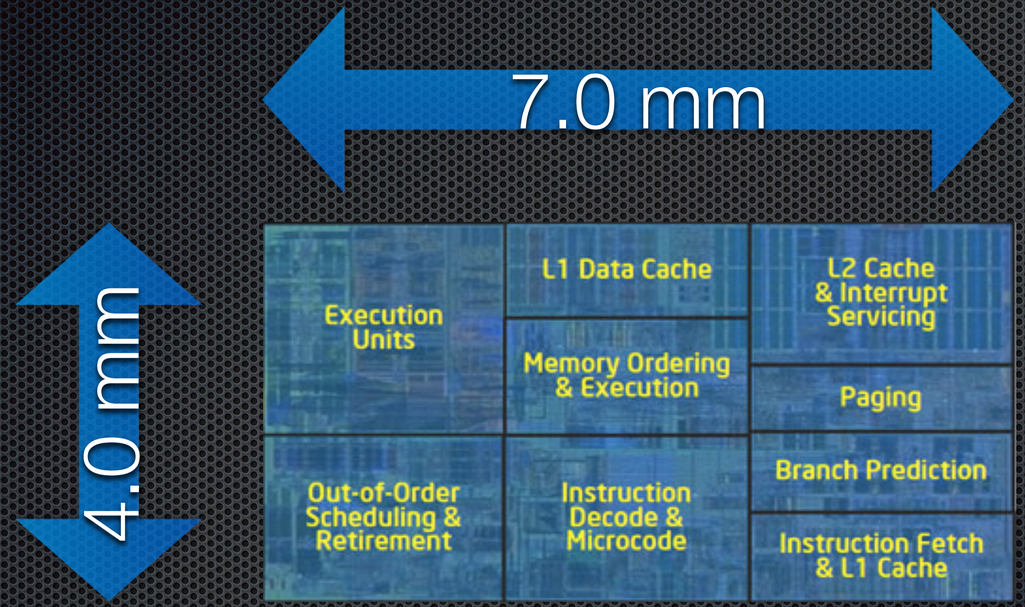
Die Dimensions



Major Block Distribution

- L3 Cache: 30%
- Cores: 45% (11.2% each)
- Memory Controller: 13% (more than one core)
- Misc. I/O: 5%
- Quickpath: 5%
- Queue: 2%

Core Dimensions

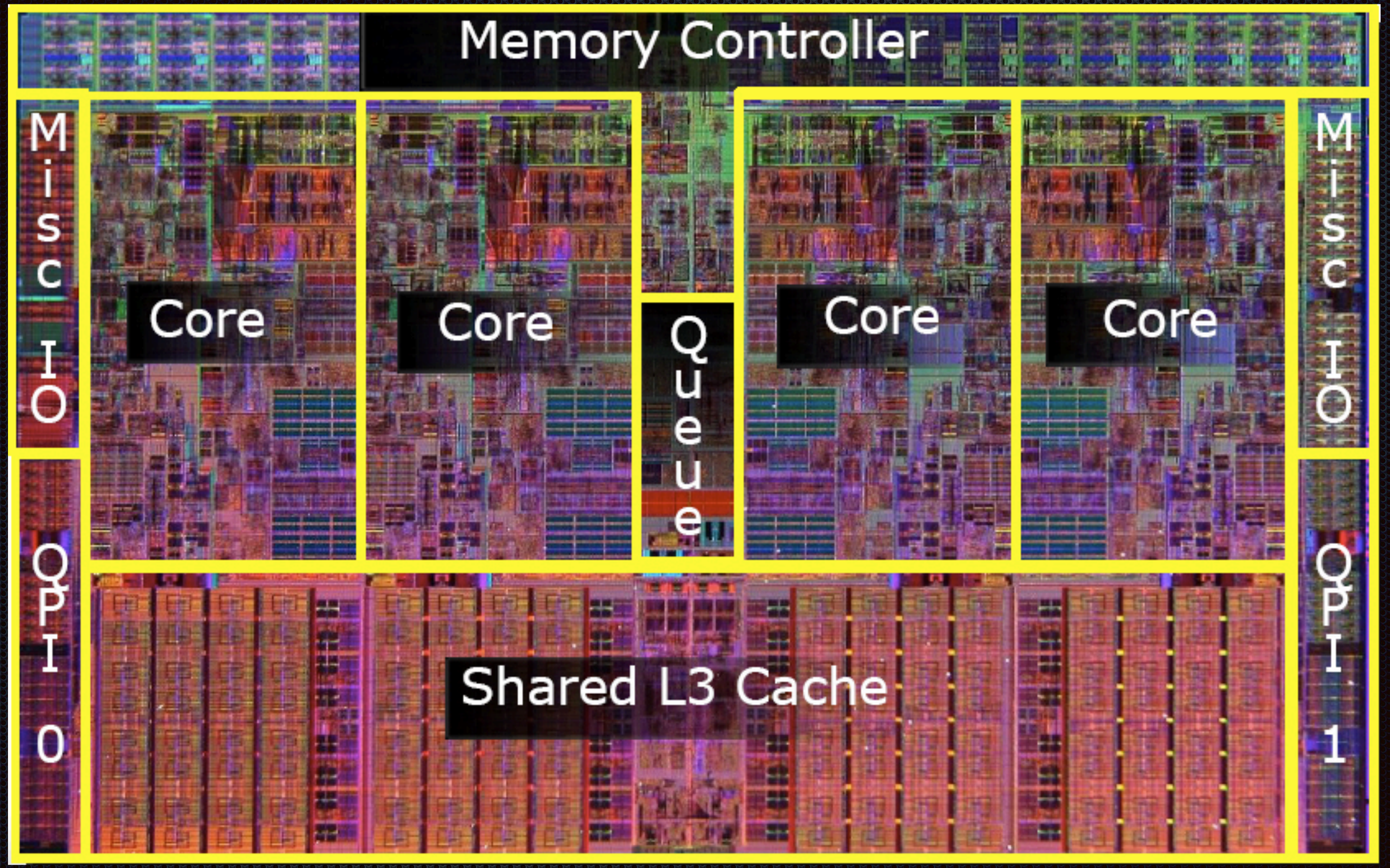


Core Area Distribution

- Execution Units: 19%
- L1 Data Cache: 8%
- L1 Instruction Cache: 9%
- L2 Cache: 10%
- Memory Ordering and Execution: 10%
- Out of order scheduling: 15%
- Paging (TLB): 6%
- Branch Prediction: 7%
- Decode: 15%

Overall Distribution

- Execution units: 8.6%
- Memory: 42% (60% including controllers)
- Instruction flow management: 17%


Memory Controller
Misc IO
Core
Core
Queue
Core
Core
Misc IO
QPI 0
Shared L3 Cache
QPI 1

Summary

- Processors are becoming “smart memory”
- Heat is concentrated in a small fraction of the chip
  - but spread out across 45% of area
- More area dedicated to external access

Why Cache?

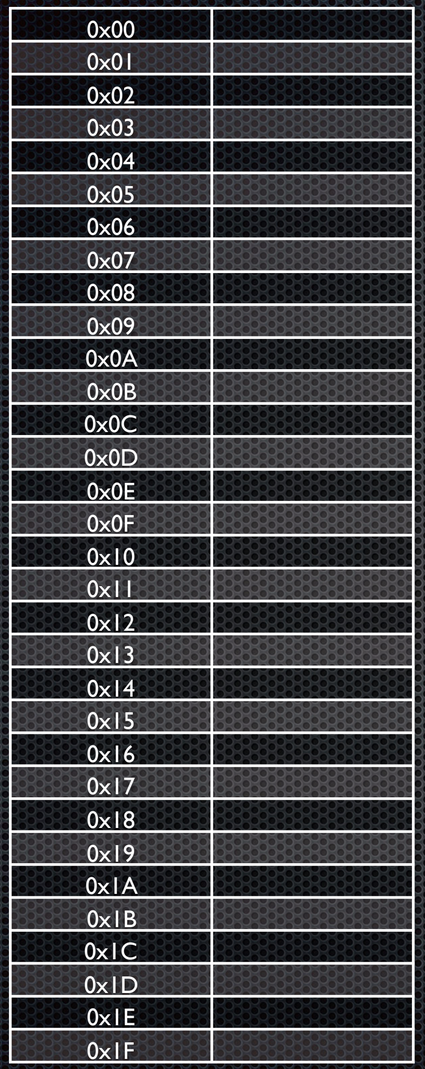
- As RAM grows in size, access time increases
- RAM is on chips external to the processor, also decreasing access time
- Cache provides a small, fast, on-chip working memory for the processor, that is backed by RAM

Why Mapping?

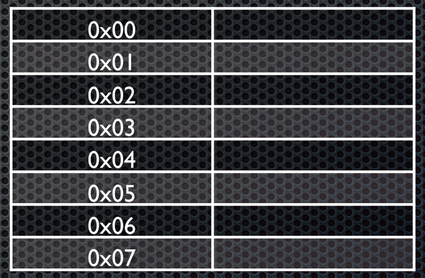
- Because cache is smaller, RAM addresses must go through a mapping function to be converted to cache addresses
- We cannot fit all of the contents of RAM into cache at once
- Any address mapping will be many-to-one

Memory Maps

RAM





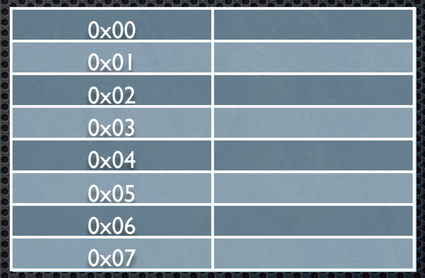


Cache

Memory Maps

RAM



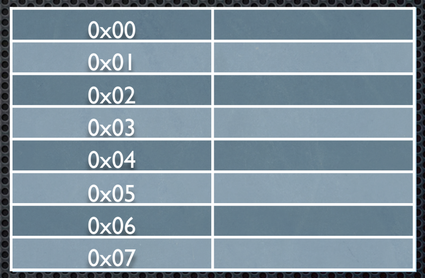


Cache

Memory Maps

RAM



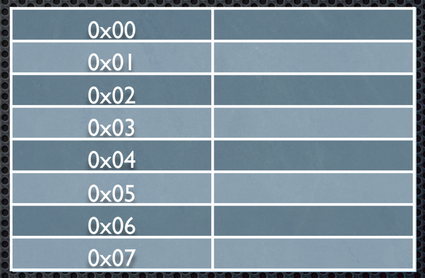


Cache

Memory Maps

RAM



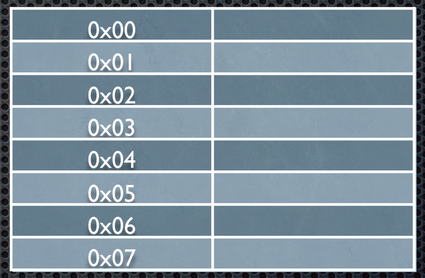


Cache

Memory Maps

RAM





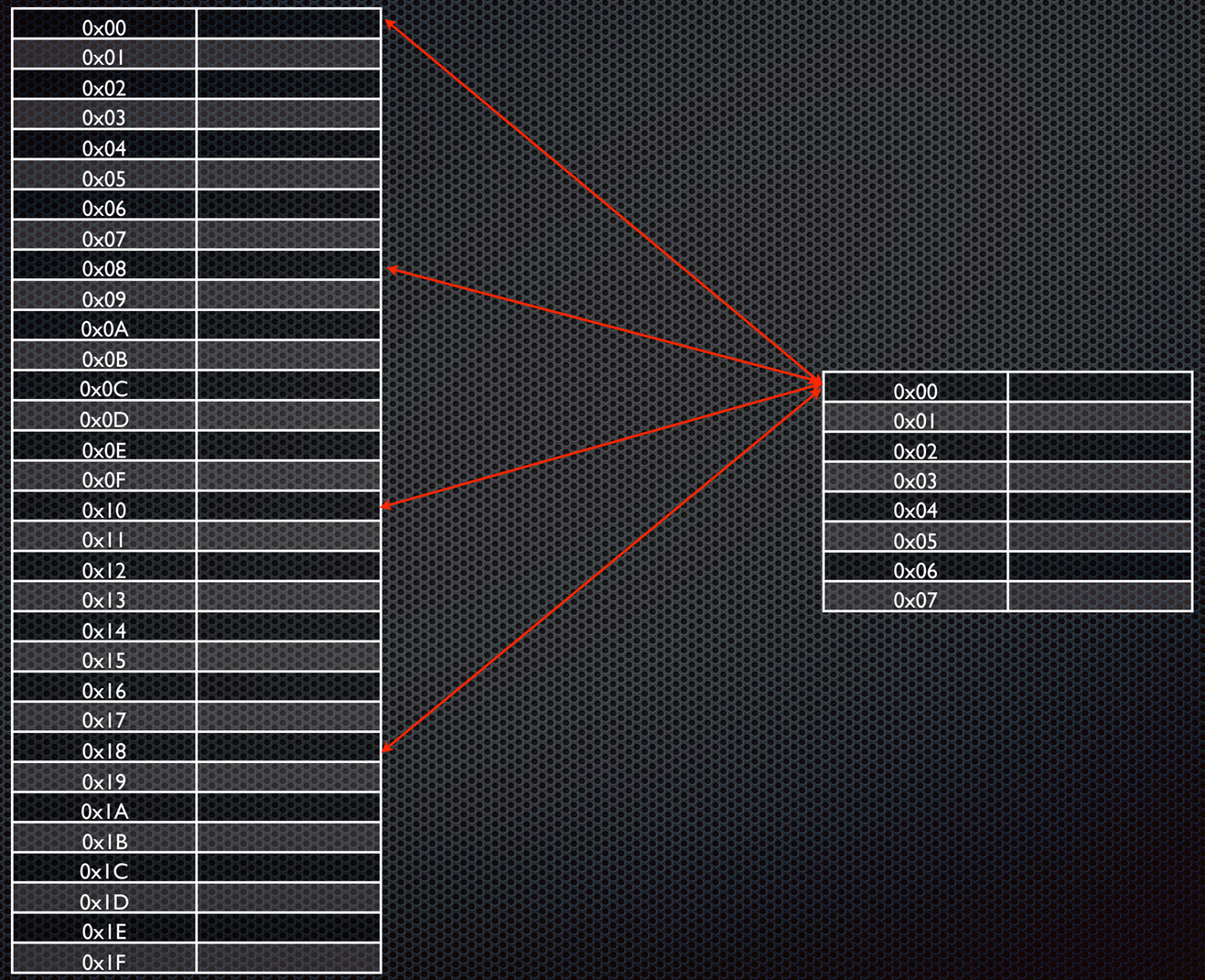
Cache

Mapping

- Each location in our 8-word cache has four corresponding locations in this 32-word RAM
- 0x00 can match 0x00, 0x08, 0x10, or 0x18
- Reversing this, we can say that if we ignore the top 2 bits of the RAM address, we have the cache address
- The lower 3 bits of the RAM address are the same as the cache address

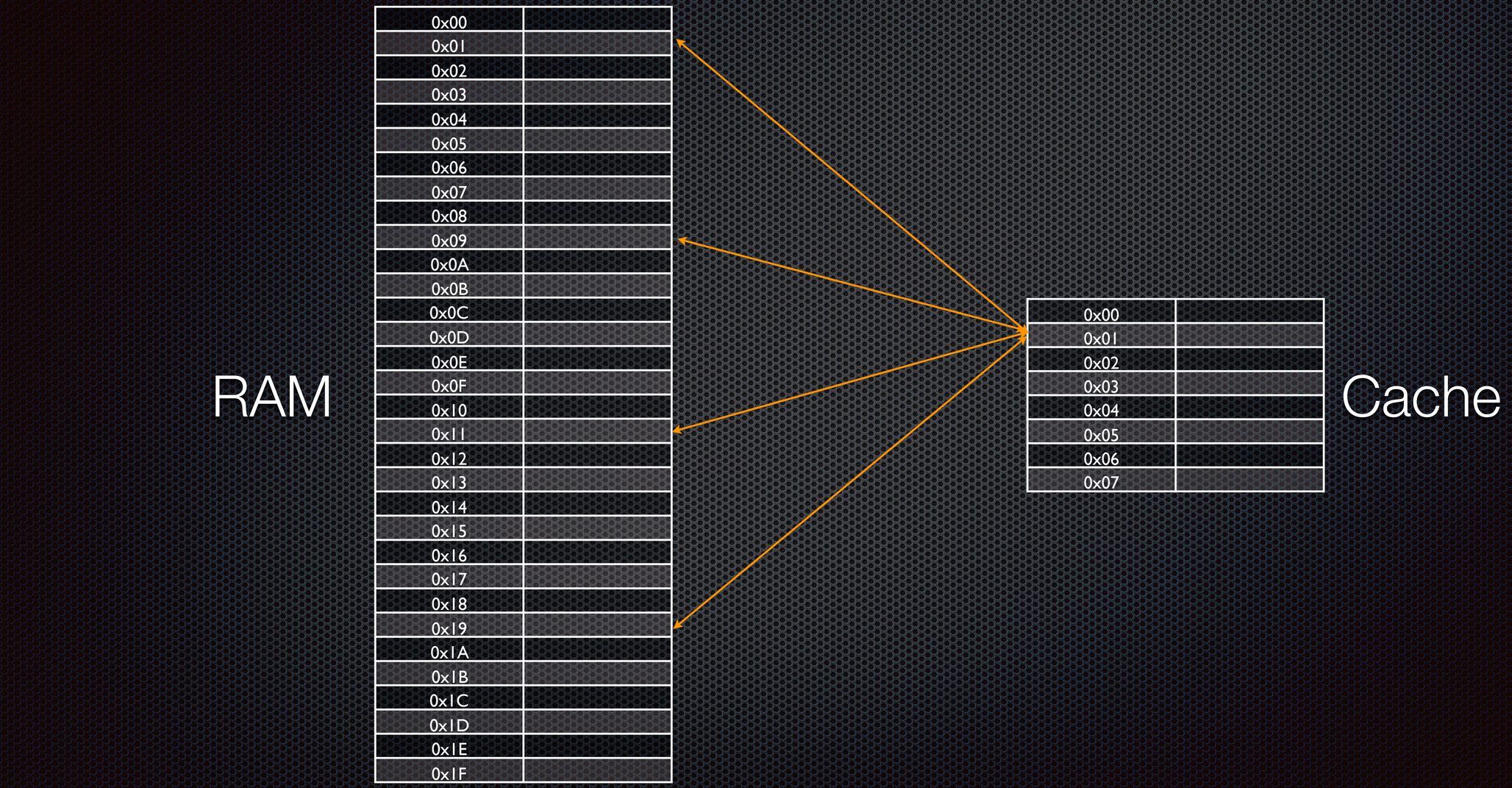
Another View

RAM



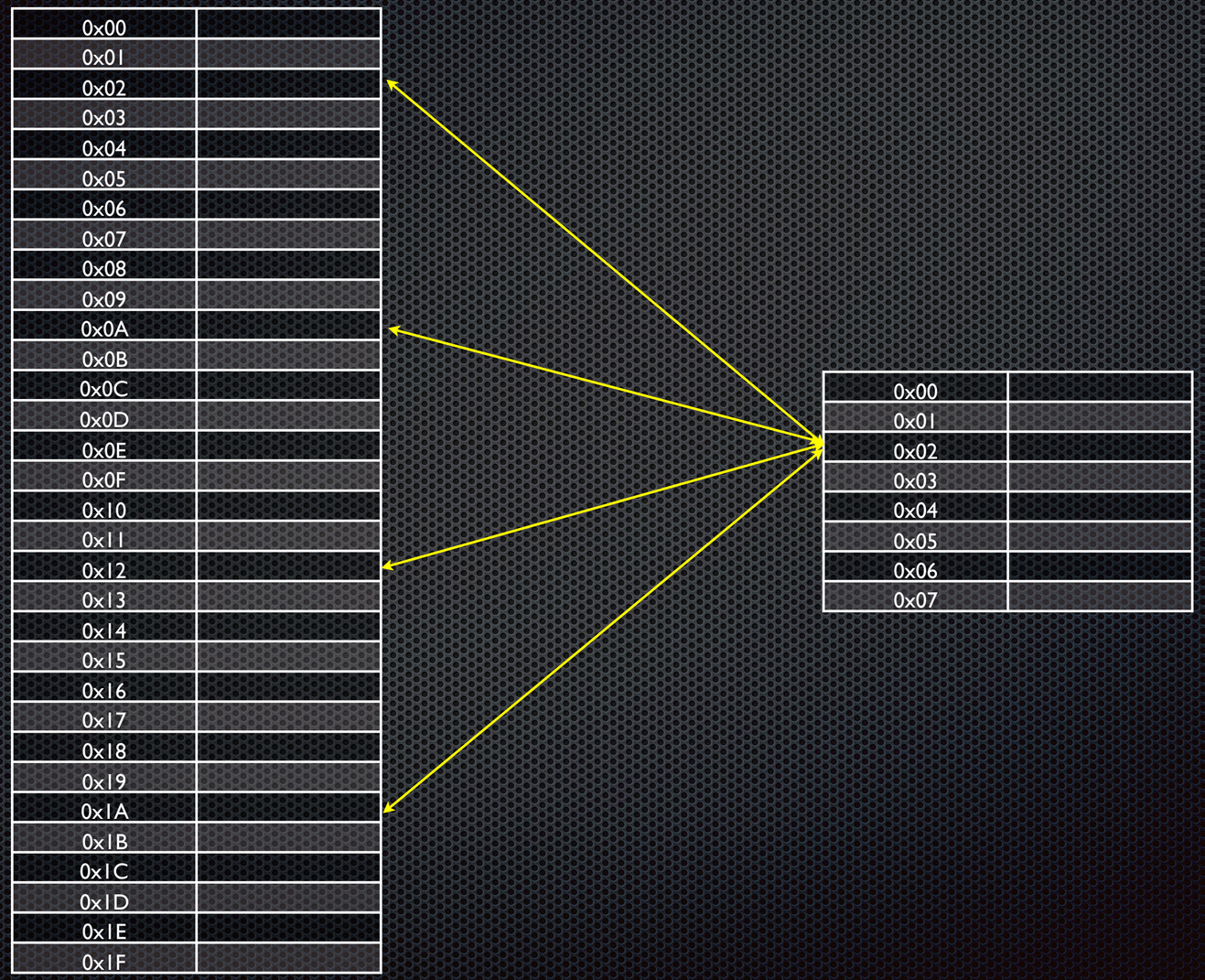
Cache

Another View



Another View

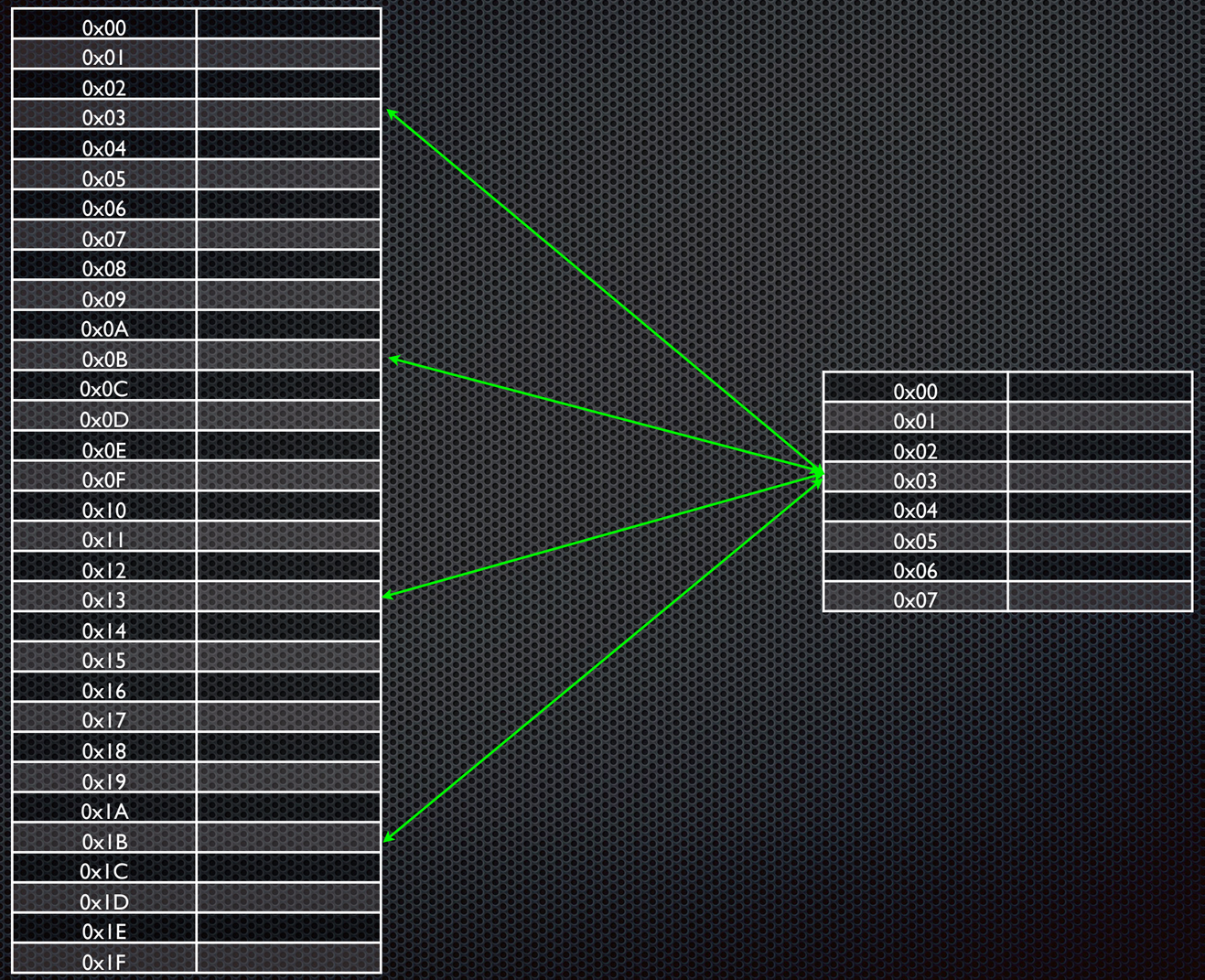
RAM



Cache

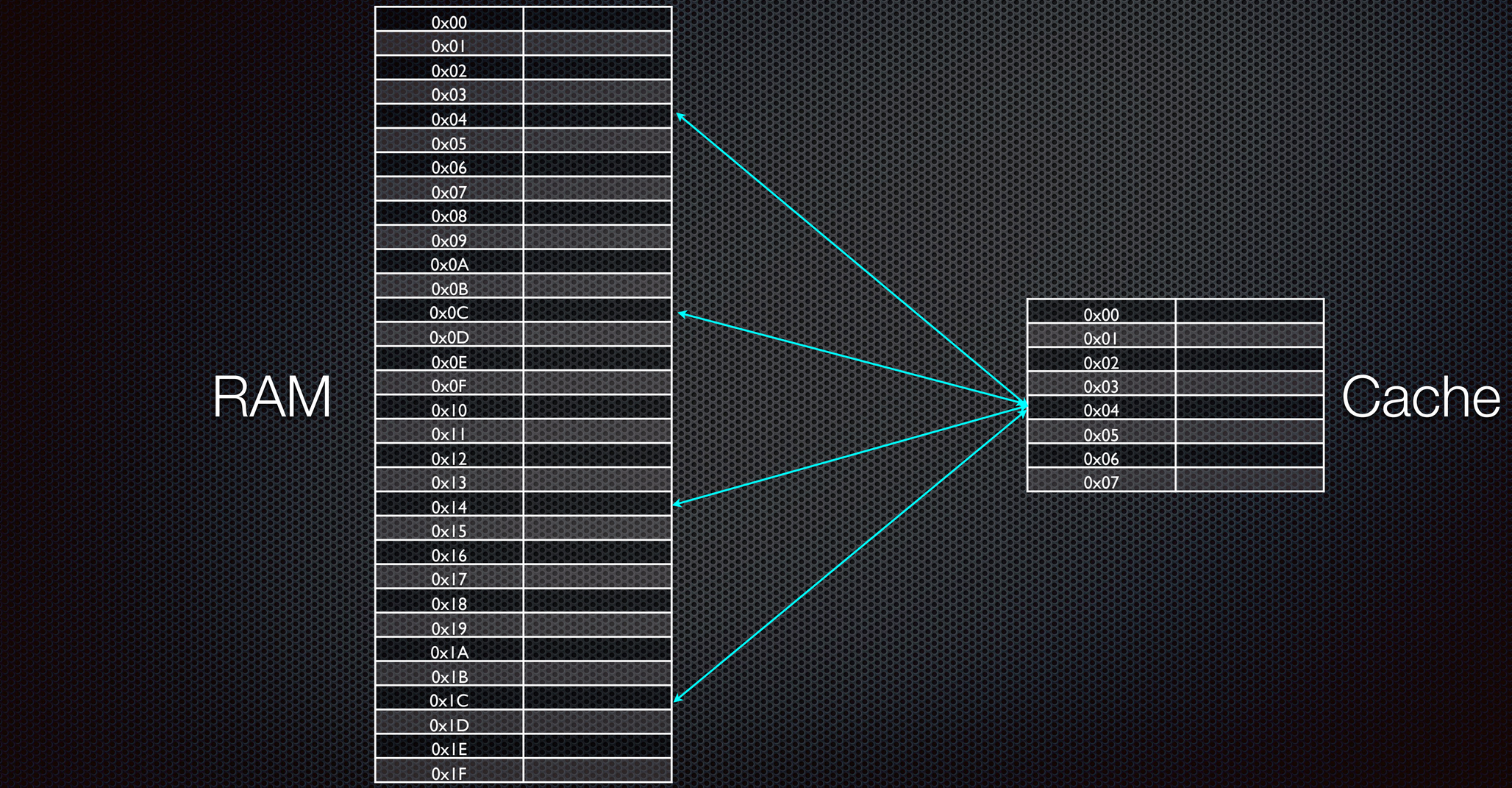
Another View

RAM



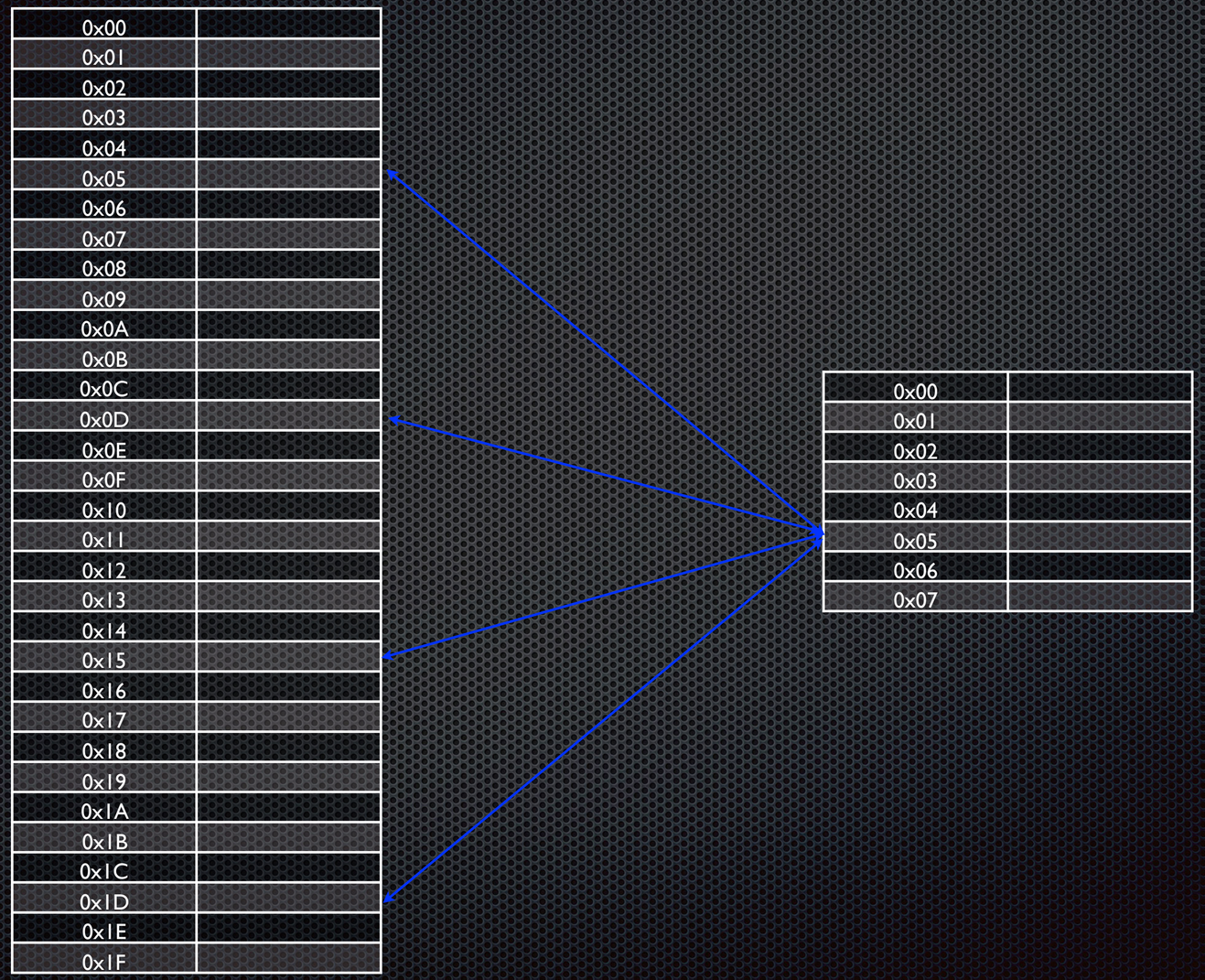
Cache

Another View



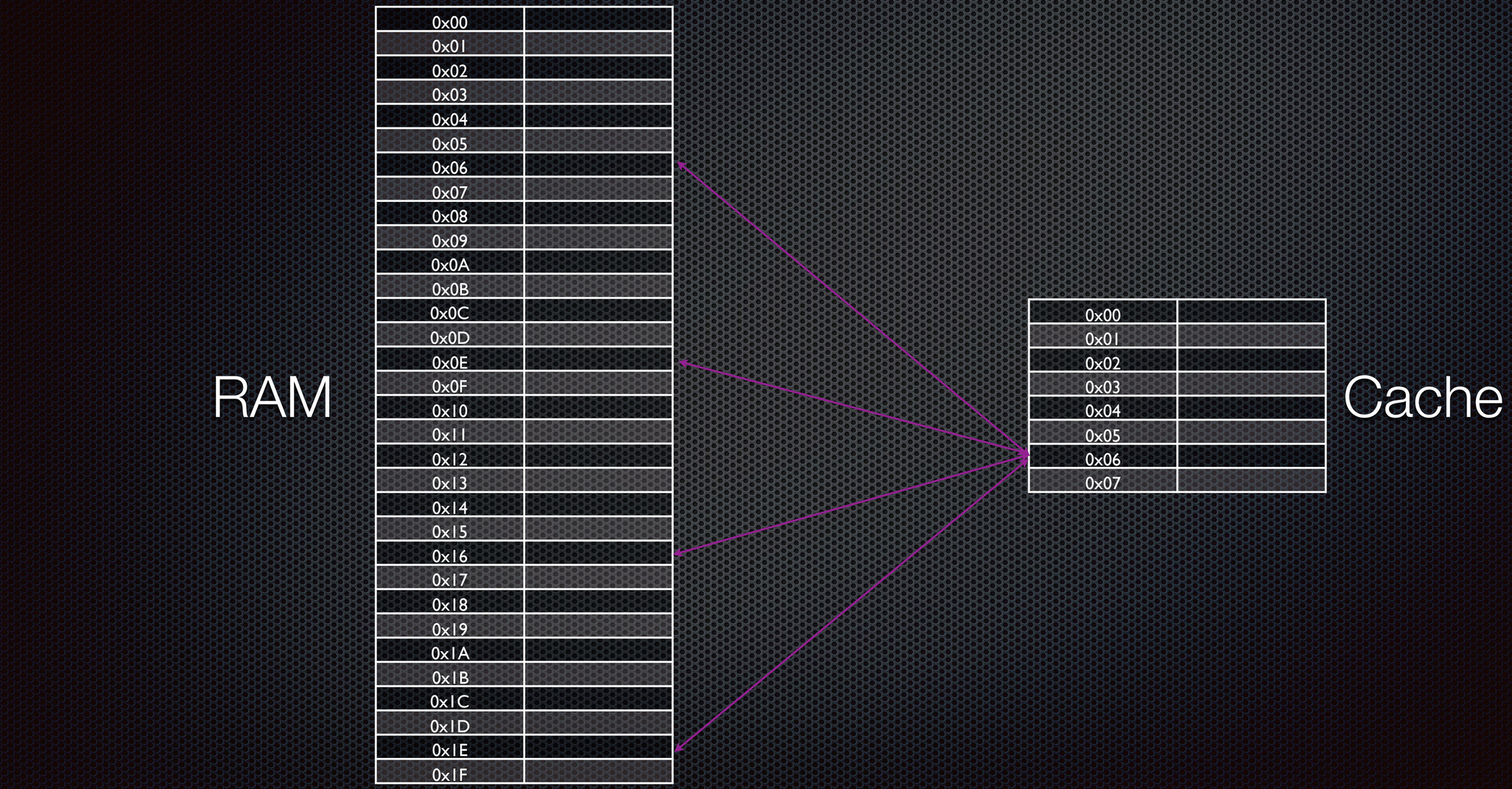
Another View

RAM



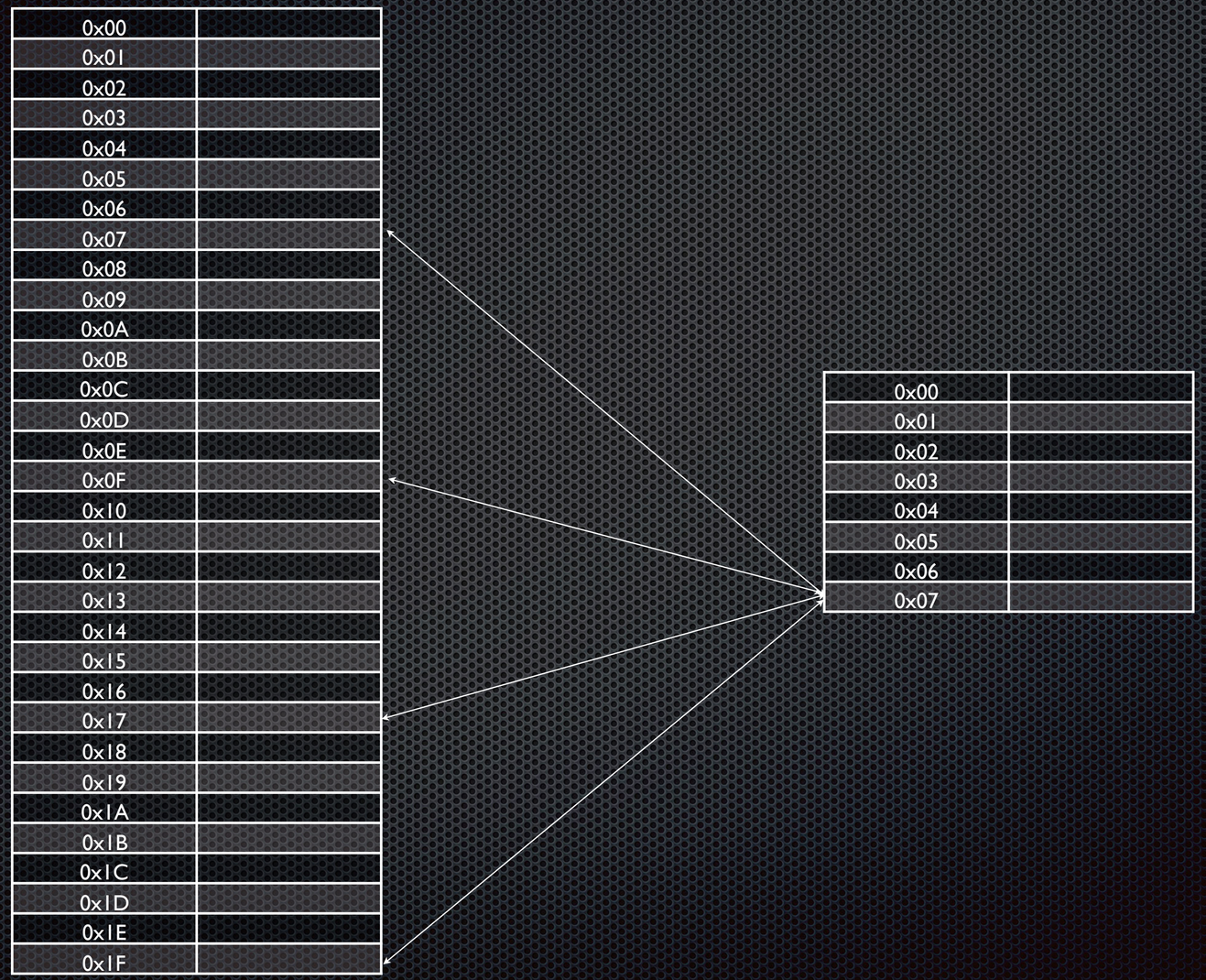
Cache

Another View



Another View

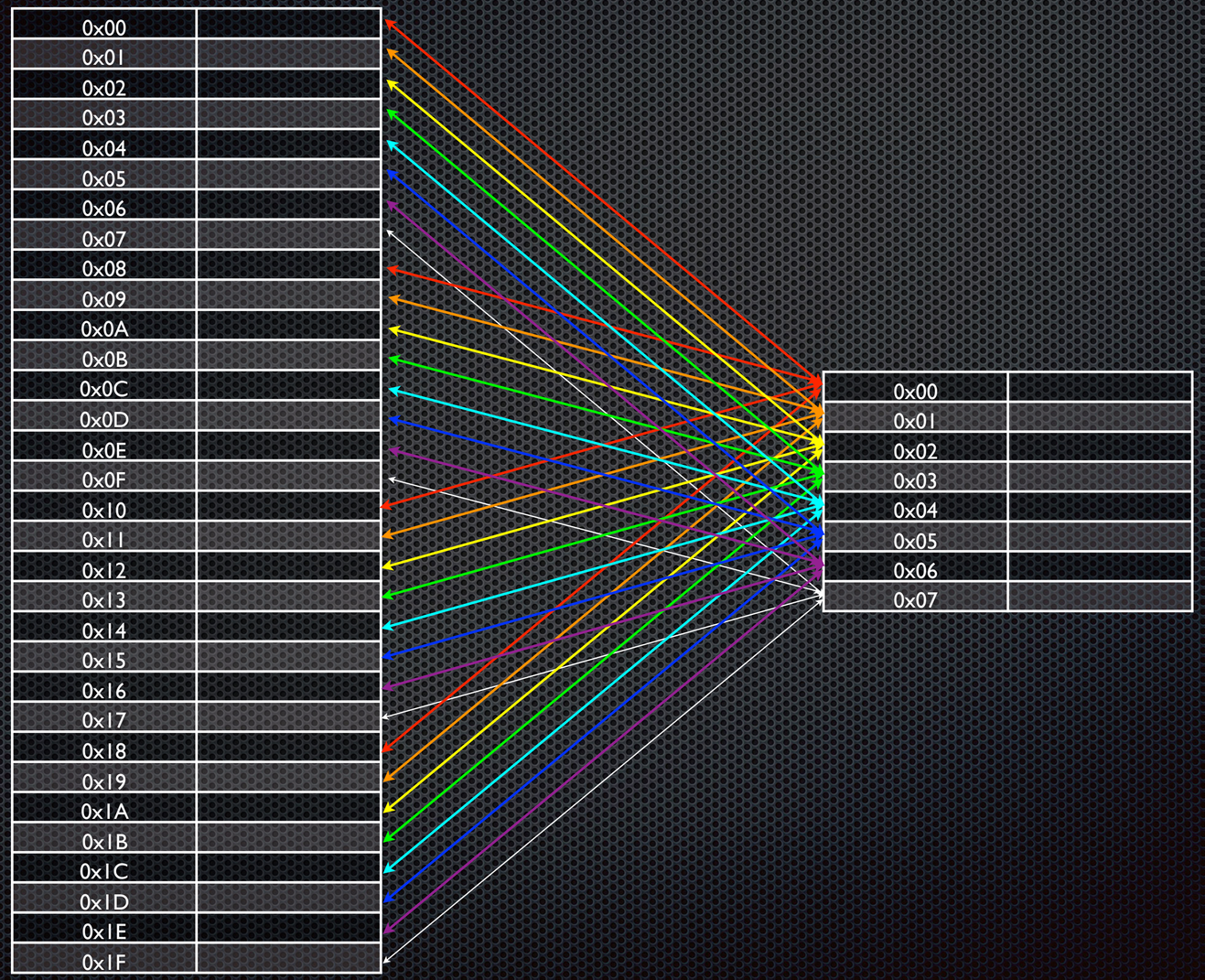
RAM



Cache

Another View

RAM



Cache

Reverse Mapping

- Once we have a RAM value in cache, how do we know which one of the four corresponding locations it came from?
- We need to extend the cache with a field that remembers the extra 2 bits of the RAM address for the current value
- We call this the cache tag field

Cache Tag*

RAM

0b00 000	
0b00 001	
0b00 010	
0b00 011	
0b00 100	
0b00 101	
0b00 110	
0b00 111	
0b01 000	
0b01 001	
0b01 010	
0b01 011	
0b01 100	
0b01 101	
0b01 110	
0b01 111	
0b10 000	
0b10 001	
0b10 010	
0b10 011	
0b10 100	
0b10 101	
0b10 110	
0b10 111	
0b11 000	
0b11 001	
0b11 010	
0b11 011	
0b11 100	
0b11 101	
0b11 110	
0b11 111	

The TAG stores the upper 2 bits of the RAM address

The INDEX is implicit -- it is just the address within the cache

Let's follow a series of read operations that load the marked words of RAM into the cache

TAG	INDEX	DATA
	0b000	
	0b001	
	0b010	
	0b011	
	0b100	
	0b101	
	0b110	
	0b111	

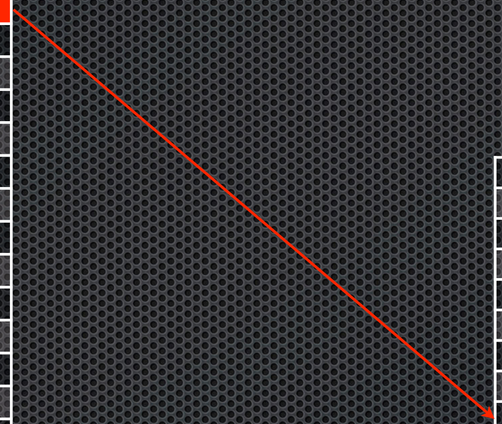
Cache

*Note switch to binary notation

Cache Tag

RAM

0b00 000	
0b00 001	
0b00 010	
0b00 011	
0b00 100	
0b00 101	
0b00 110	
0b00 111	
0b01 000	
0b01 001	
0b01 010	
0b01 011	
0b01 100	
0b01 101	
0b01 110	
0b01 111	
0b10 000	
0b10 001	
0b10 010	
0b10 011	
0b10 100	
0b10 101	
0b10 110	
0b10 111	
0b11 000	
0b11 001	
0b11 010	
0b11 011	
0b11 100	
0b11 101	
0b11 110	
0b11 111	



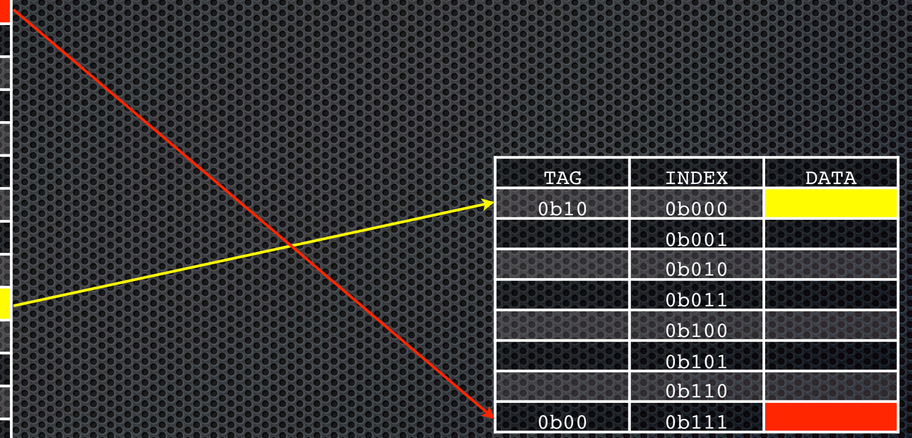
TAG	INDEX	DATA
0b10	0b000	
	0b001	
0b01	0b010	
0b10	0b011	
	0b100	
0b11	0b101	
	0b110	
0b00	0b111	

Cache

Cache Tag

RAM

0b00 000	
0b00 001	
0b00 010	
0b00 011	
0b00 100	
0b00 101	
0b00 110	
0b00 111	
0b01 000	
0b01 001	
0b01 010	
0b01 011	
0b01 100	
0b01 101	
0b01 110	
0b01 111	
0b10 000	
0b10 001	
0b10 010	
0b10 011	
0b10 100	
0b10 101	
0b10 110	
0b10 111	
0b11 000	
0b11 001	
0b11 010	
0b11 011	
0b11 100	
0b11 101	
0b11 110	
0b11 111	



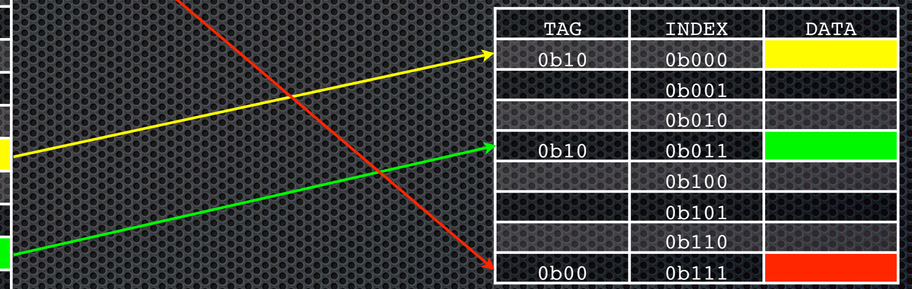
TAG	INDEX	DATA
0b10	0b000	
	0b001	
	0b010	
	0b011	
	0b100	
	0b101	
	0b110	
0b00	0b111	

Cache

Cache Tag

RAM

0b00 000
0b00 001
0b00 010
0b00 011
0b00 100
0b00 101
0b00 110
0b00 111
0b01 000
0b01 001
0b01 010
0b01 011
0b01 100
0b01 101
0b01 110
0b01 111
0b10 000
0b10 001
0b10 010
0b10 011
0b10 100
0b10 101
0b10 110
0b10 111
0b11 000
0b11 001
0b11 010
0b11 011
0b11 100
0b11 101
0b11 110
0b11 111



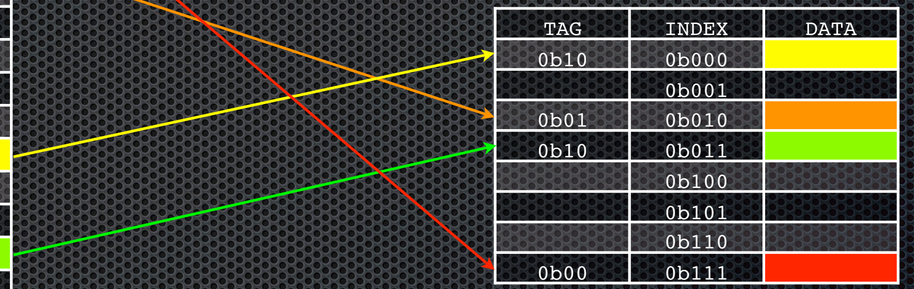
TAG	INDEX	DATA
0b10	0b000	
	0b001	
	0b010	
0b10	0b011	
	0b100	
	0b101	
	0b110	
0b00	0b111	

Cache

Cache Tag

RAM

0b00 000
0b00 001
0b00 010
0b00 011
0b00 100
0b00 101
0b00 110
0b00 111
0b01 000
0b01 001
0b01 010
0b01 011
0b01 100
0b01 101
0b01 110
0b01 111
0b10 000
0b10 001
0b10 010
0b10 011
0b10 100
0b10 101
0b10 110
0b10 111
0b11 000
0b11 001
0b11 010
0b11 011
0b11 100
0b11 101
0b11 110
0b11 111



TAG	INDEX	DATA
0b10	0b000	
	0b001	
0b01	0b010	
0b10	0b011	
	0b100	
	0b101	
	0b110	
0b00	0b111	

Cache

Cache Tag



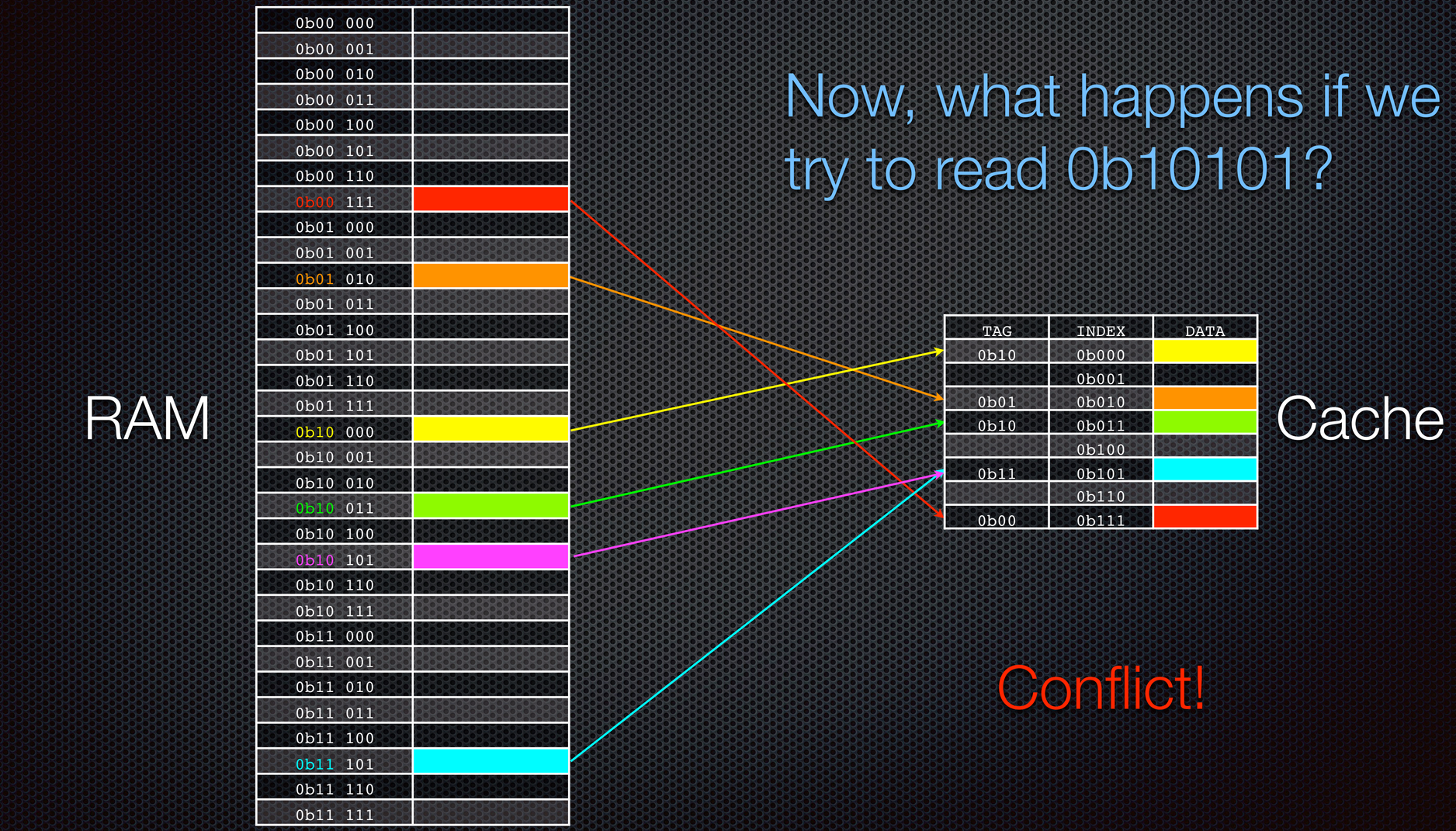
RAM

Now, what happens if we try to read 0b10101?

TAG	INDEX	DATA
0b10	0b000	
	0b001	
0b01	0b010	
0b10	0b011	
	0b100	
0b11	0b101	
	0b110	
0b00	0b111	

Cache

Cache Tag



Now, what happens if we try to read 0b10101?

Conflict!

Conflict Resolution

RAM



The new value must replace the existing value in cache

TAG	INDEX	DATA
0b10	0b000	
	0b001	
0b01	0b010	
0b10	0b011	
	0b100	
0b11	0b101	
	0b110	
0b00	0b111	

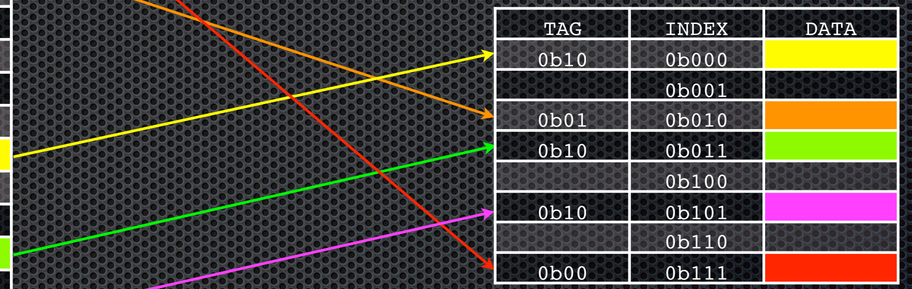
Cache

If the cache value hasn't changed, this is fine

Conflict Resolution

RAM

0b00 000	
0b00 001	
0b00 010	
0b00 011	
0b00 100	
0b00 101	
0b00 110	
0b00 111	
0b01 000	
0b01 001	
0b01 010	
0b01 011	
0b01 100	
0b01 101	
0b01 110	
0b01 111	
0b10 000	
0b10 001	
0b10 010	
0b10 011	
0b10 100	
0b10 101	
0b10 110	
0b10 111	
0b11 000	
0b11 001	
0b11 010	
0b11 011	
0b11 100	
0b11 101	
0b11 110	
0b11 111	



Cache

The new value must replace the existing value in cache

If the cache value hasn't changed, this is fine

Eviction

RAM



If the value in cache has changed, then it must first be written back to RAM

TAG	INDEX	DATA
0b10	0b000	
	0b001	
0b01	0b010	
0b10	0b011	
	0b100	
0b11	0b101	
	0b110	
0b00	0b111	

Cache

Eviction

RAM



If the value in cache has changed, then it must first be written back to RAM

TAG	INDEX	DATA
0b10	0b000	
	0b001	
0b01	0b010	
0b10	0b011	
	0b100	
0b11	0b101	
	0b110	
0b00	0b111	

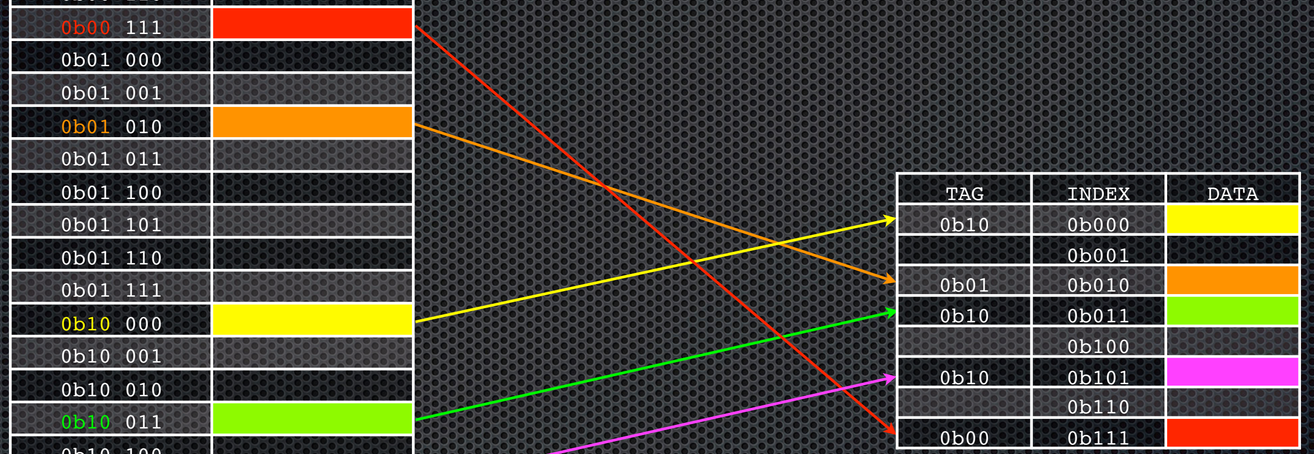
Cache

Then the new value can take its place

Replacement

RAM

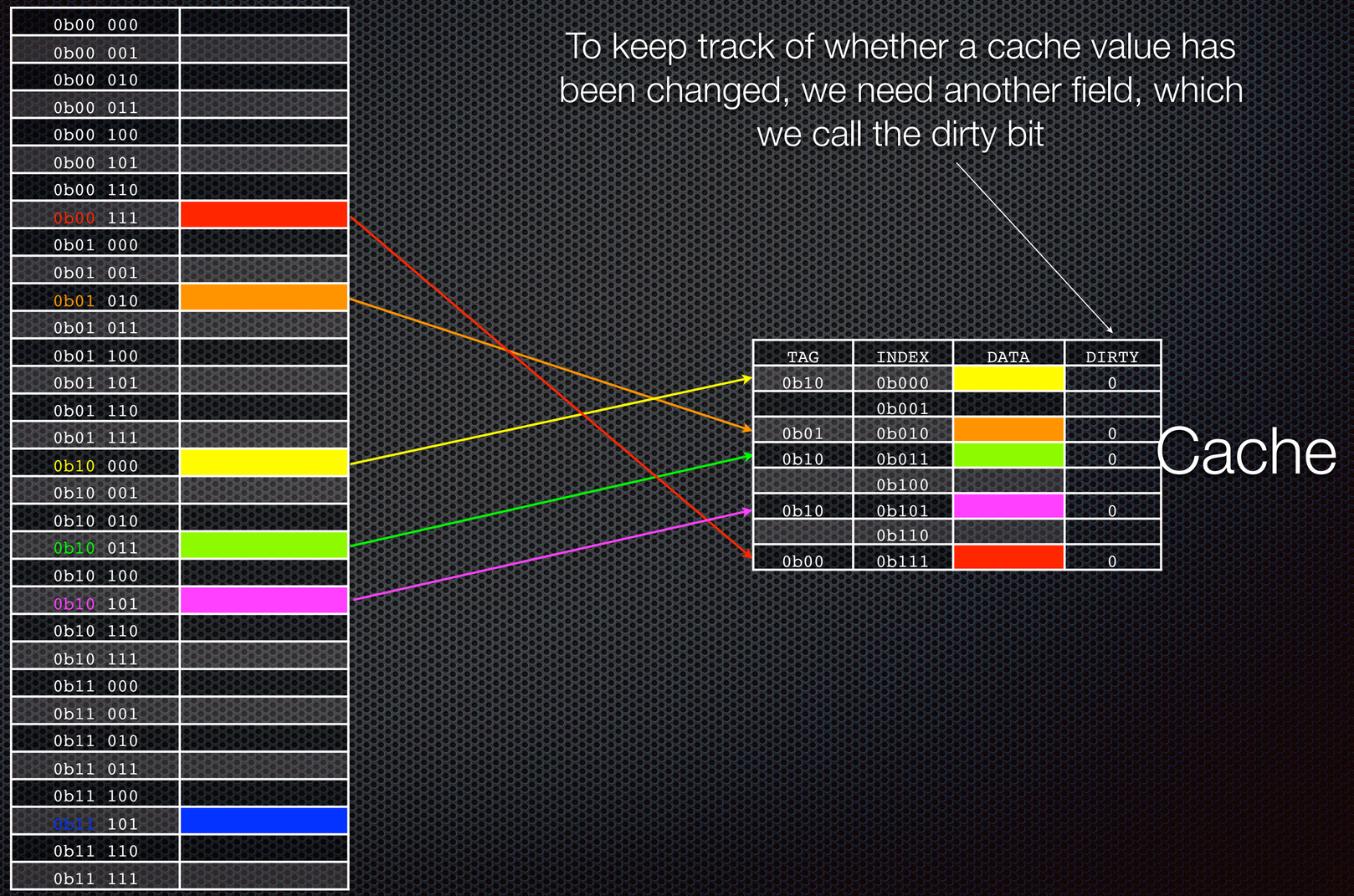
0b00 000	
0b00 001	
0b00 010	
0b00 011	
0b00 100	
0b00 101	
0b00 110	
0b00 111	
0b01 000	
0b01 001	
0b01 010	
0b01 011	
0b01 100	
0b01 101	
0b01 110	
0b01 111	
0b10 000	
0b10 001	
0b10 010	
0b10 011	
0b10 100	
0b10 101	
0b10 110	
0b10 111	
0b11 000	
0b11 001	
0b11 010	
0b11 011	
0b11 100	
0b11 101	
0b11 110	
0b11 111	



TAG	INDEX	DATA
0b10	0b000	
	0b001	
0b01	0b010	
0b10	0b011	
	0b100	
0b10	0b101	
	0b110	
0b00	0b111	

Cache

Dirty Bit


0b00 000
0b00 001
0b00 010
0b00 011
0b00 100
0b00 101
0b00 110
0b00 111
0b01 000
0b01 001
0b01 010
0b01 011
0b01 100
0b01 101
0b01 110
0b01 111
0b10 000
0b10 001
0b10 010
0b10 011
0b10 100
0b10 101
0b10 110
0b10 111
0b11 000
0b11 001
0b11 010
0b11 011
0b11 100
0b11 101
0b11 110
0b11 111
RAM
To keep track of whether a cache value has been changed, we need another field, which we call the dirty bit
TAG
INDEX
DATA
DIRTY
0b10
0b000
0
0b001
0b01
0b010
0
0b10
0b011
0
0b100
0b10
0b101
0
0b110
0b00
0b111
0
Cache

Writing

- Writing from the processor to memory presents design options:
  - Should we write to cache, RAM, or both?

Write Policy

- Cache only: Write-Back, Allocate
  - Evicts dirty cache data, then is stored with dirty bit set
- RAM only: Write-Through, No-allocate
  - Does not evict cache, must be subsequently read to enter cache
- Cache & RAM: Write-Through, Allocate
  - Evicts dirty cache data, then is stored in RAM and cache, dirty bit is not set

Locality

Deja Vu All Over Again

Temporal Locality

- Our simple cache exploits a statistical property of common programs:
  - There is a higher probability that we will access a location we've recently accessed than some other random location in the entire address space
  - Put another way, if RAM has a billion words, which would you bet on a program accessing next? An address it has been working with recently, or one chosen randomly out of a billion?

Spatial Locality

- But what if that “random” location happens to immediately follow one it just read?
- That might be a pretty good bet too! (Especially if the PC or an index register is pointing to it)
- We can exploit this by reading a group of words into cache, whenever we need to bring in one new value

Cache Lines

RAM

0b0 000 0
0b0 000 1
0b0 001 0
0b0 001 1
0b0 010 0
0b0 010 1
0b0 011 0
0b0 011 1
0b0 100 0
0b0 100 1
0b0 101 0
0b0 101 1
0b0 110 0
0b0 110 1
0b0 111 0
0b0 111 1
0b1 000 0
0b1 000 1
0b1 001 0
0b1 001 1
0b1 010 0
0b1 010 1
0b1 011 0
0b1 011 1
0b1 100 0
0b1 100 1
0b1 101 0
0b1 101 1
0b1 110 0
0b1 110 1
0b1 111 0
0b1 111 1

There are several important points to note about this cache organization:

TAG	INDEX	DATA 1	DATA 0	D
0b1	0b000			
0b1	0b001			
	0b010			
0b0	0b011			
	0b100			
0b0	0b101			
	0b110			
0b1	0b111			

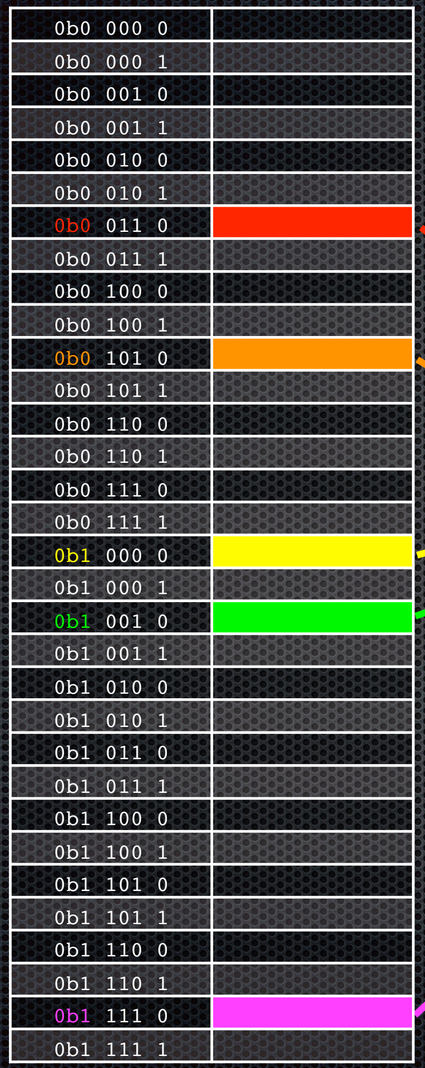
Cache

It’s twice as big

Cache Lines

There are several important points to note about this cache organization:

RAM

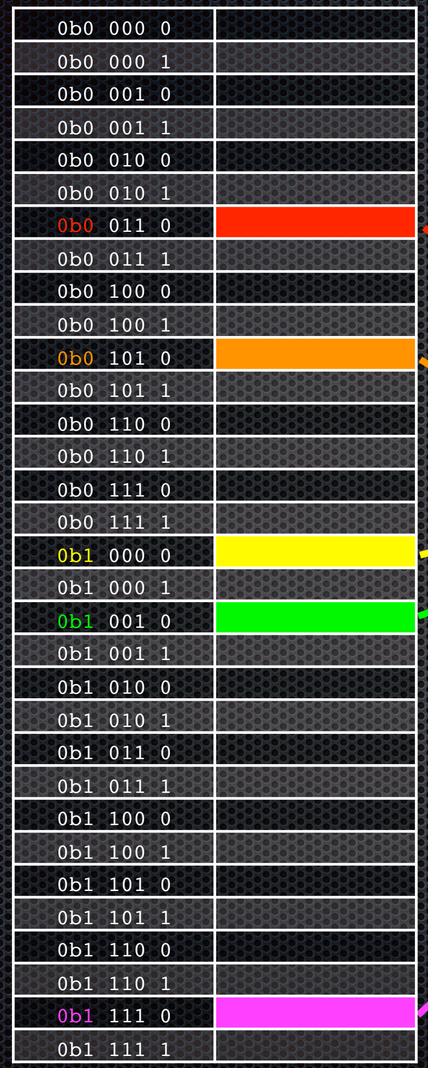


Cache

TAG	INDEX	DATA 1	DATA 0	D
0b1	0b000			
0b1	0b001			
	0b010			
0b0	0b011			
	0b100			
0b0	0b101			
	0b110			
0b0	0b111			

Each word within the line is selected by its offset

Cache Lines



RAM

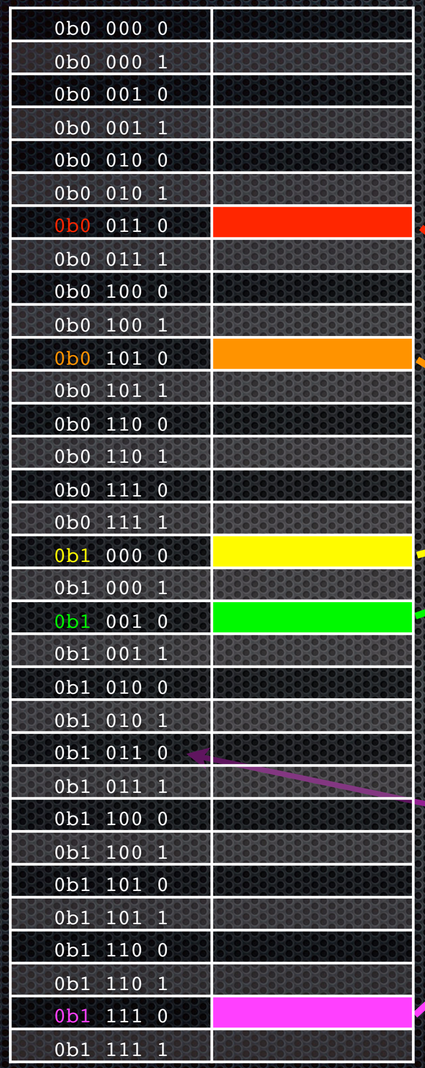
There are several important points to note about this cache organization:

TAG	INDEX	DATA 1	DATA 0	D
0b1	0b000			
0b1	0b001			
	0b010			
0b0	0b011			
	0b100			
0b0	0b101			
	0b110			
0b0	0b111			

Cache

The tag is now just one bit

Cache Lines



RAM

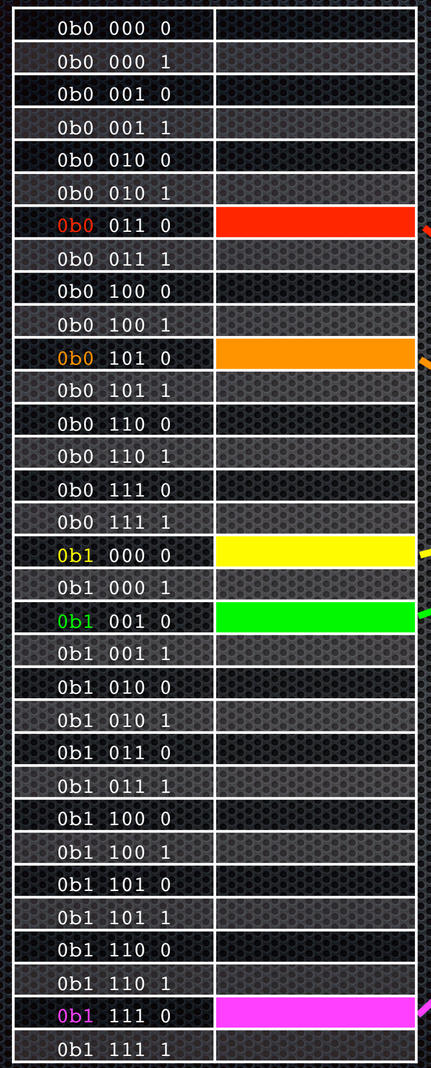
There are several important points to note about this cache organization:

TAG	INDEX	DATA 1	DATA 0	D
0b1	0b000			
0b1	0b001			
	0b010			
0b0	0b011			
	0b100			
0b0	0b101			
	0b110			
0b0	0b111			

Cache

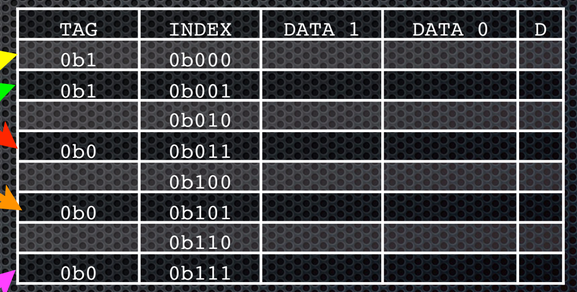
The RAM address is logically divided into Tag, Index, and Offset

Cache Lines



RAM

There are several important points to note about this cache organization:



TAG	INDEX	DATA 1	DATA 0	D
0b1	0b000			
0b1	0b001			
	0b010			
0b0	0b011			
	0b100			
0b0	0b101			
	0b110			
0b0	0b111			

Cache

Each line holds two values

Cache Lines

RAM

0b0 000 0
0b0 000 1
0b0 001 0
0b0 001 1
0b0 010 0
0b0 010 1
0b0 011 0
0b0 011 1
0b0 100 0
0b0 100 1
0b0 101 0
0b0 101 1
0b0 110 0
0b0 110 1
0b0 111 0
0b0 111 1
0b1 000 0
0b1 000 1
0b1 001 0
0b1 001 1
0b1 010 0
0b1 010 1
0b1 011 0
0b1 011 1
0b1 100 0
0b1 100 1
0b1 101 0
0b1 101 1
0b1 110 0
0b1 110 1
0b1 111 0
0b1 111 1

There are several important points to note about this cache organization:

TAG	INDEX	DATA 1	DATA 0	D
0b1	0b000			
0b1	0b001			
	0b010			
0b0	0b011			
	0b100			
0b0	0b101			
	0b110			
0b0	0b111			

Cache

Each pair of values in cache comes from two consecutive locations in RAM

Cache Lines

RAM

0b0 000 0
0b0 000 1
0b0 001 0
0b0 001 1
0b0 010 0
0b0 010 1
0b0 011 0
0b0 011 1
0b0 100 0
0b0 100 1
0b0 101 0
0b0 101 1
0b0 110 0
0b0 110 1
0b0 111 0
0b0 111 1
0b1 000 0
0b1 000 1
0b1 001 0
0b1 001 1
0b1 010 0
0b1 010 1
0b1 011 0
0b1 011 1
0b1 100 0
0b1 100 1
0b1 101 0
0b1 101 1
0b1 110 0
0b1 110 1
0b1 111 0
0b1 111 1

There are several important points to note about this cache organization:

Cache

TAG	INDEX	DATA 1	DATA 0	D
0b1	0b000			
0b1	0b001			
	0b010			
0b0	0b011			
	0b100			
0b0	0b101			
	0b110			
0b0	0b111			

Each pair of values in RAM begins on an even address

Cache Lines

RAM

0b0 000 0
0b0 000 1
0b0 001 0
0b0 001 1
0b0 010 0
0b0 010 1
0b0 011 0
0b0 011 1
0b0 100 0
0b0 100 1
0b0 101 0
0b0 101 1
0b0 110 0
0b0 110 1
0b0 111 0
0b0 111 1
0b1 000 0
0b1 000 1
0b1 001 0
0b1 001 1
0b1 010 0
0b1 010 1
0b1 011 0
0b1 011 1
0b1 100 0
0b1 100 1
0b1 101 0
0b1 101 1
0b1 110 0
0b1 110 1
0b1 111 0
0b1 111 1

There are several important points to note about this cache organization:

TAG	INDEX	DATA 1	DATA 0	D
0b1	0b000			
0b1	0b001			
	0b010			
0b0	0b011			
	0b100			
0b0	0b101			
	0b110			
0b0	0b111			

Cache

Whichever value of the pair is requested by the processor, both are brought to the cache at once

Writing Lines

- When a line is in the cache, write makes it dirty. Eviction writes back both values
- When a line is not in the cache
  - No-allocate just writes back to RAM
  - Write-allocate must first read the words from memory, then update the one and set the dirty bit.

Longer Lines

TAG	INDEX	DATA 11	DATA 10	DATA 01	DATA 00	D
	0b00					
	0b01					
	0b10					
	0b11					

Greater emphasis on spatial locality

Reduces temporal locality if cache is kept same size

Associative Caches

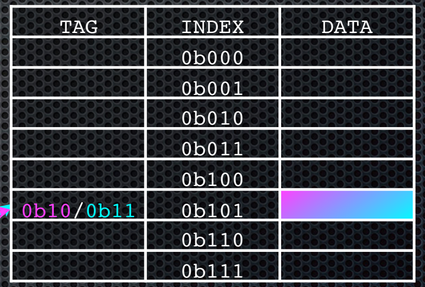
Reducing Conflicts, Increasing Utilization

Direct-Map Problem



RAM

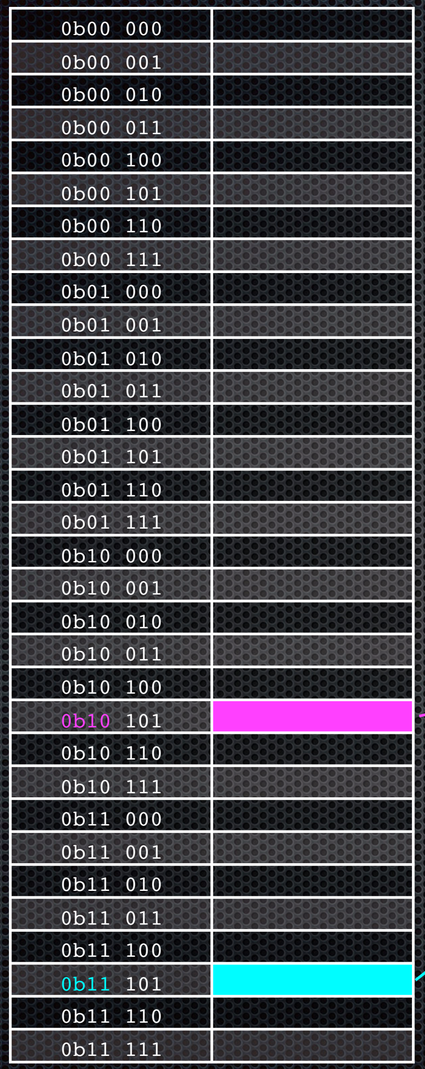
What happens if our program is in a loop, accessing 0b11101 and 0b10101?



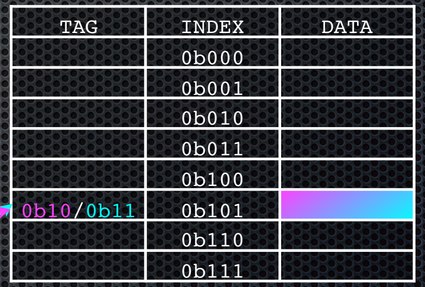
Cache

Direct-Map Problem

RAM



What happens if our program is in a loop, accessing 0b11101 and 0b10101?



Cache

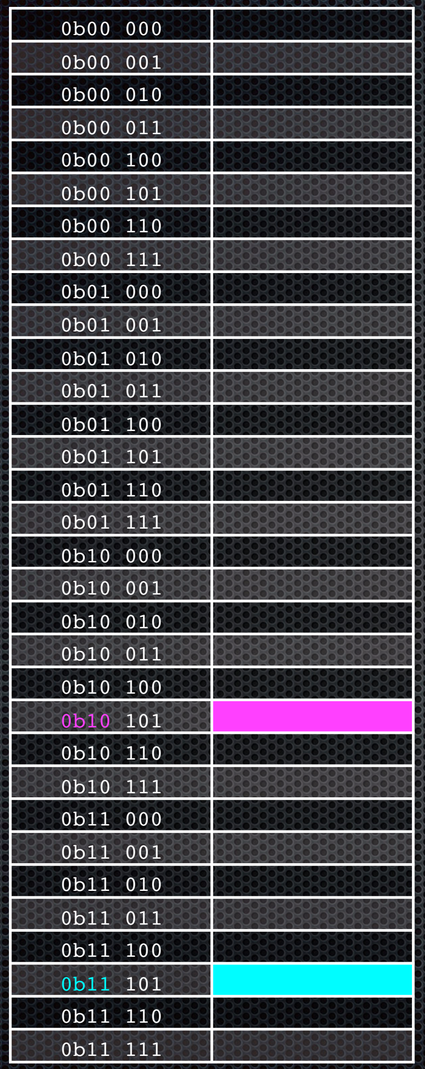
Continuous conflict:
hit rate = 0
even though the rest of
the cache is empty

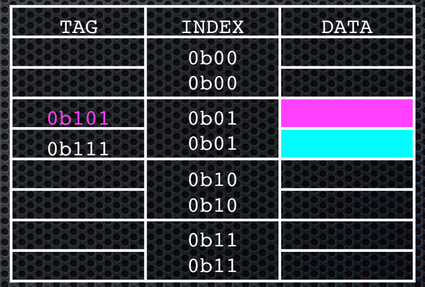
What we want is a way to share
a cache line index between two locations

Sharing means that multiple locations map to the same index, and there are multiple lines that can store memory values with the same index

What changed?

RAM

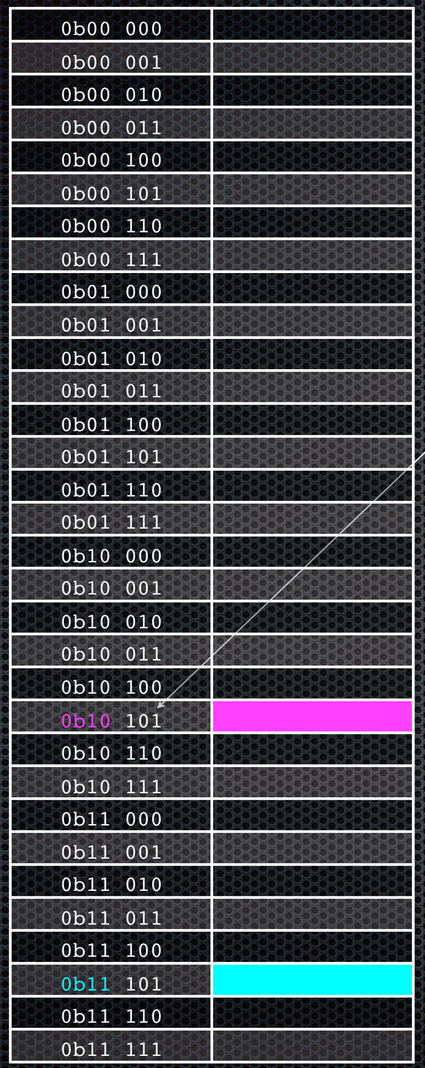




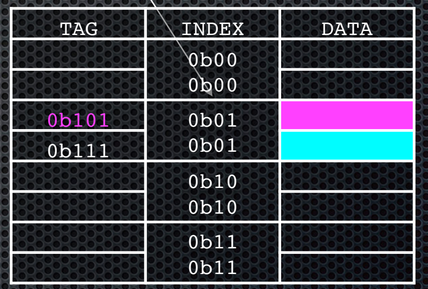
Cache

What changed?

RAM

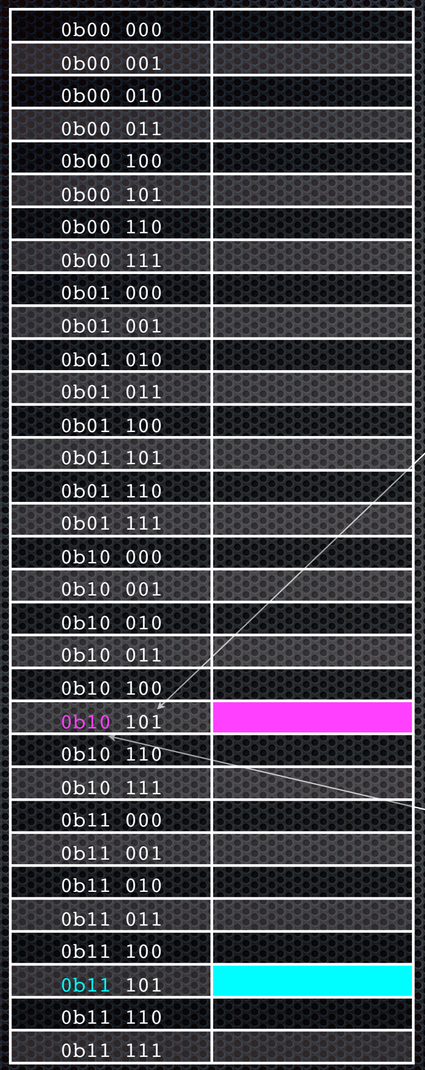


Index is one bit smaller



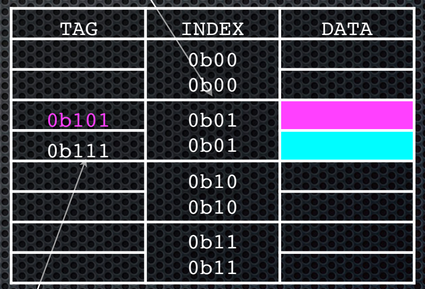
Cache

What changed?



RAM

Index is one bit smaller



Cache

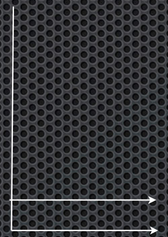
Tag is one bit bigger

What changed?

RAM

0b00 000
0b00 001
0b00 010
0b00 011
0b00 100
0b00 101
0b00 110
0b00 111
0b01 000
0b01 001
0b01 010
0b01 011
0b01 100
0b01 101
0b01 110
0b01 111
0b10 000
0b10 001
0b10 010
0b10 011
0b10 100
0b10 101
0b10 110
0b10 111
0b11 000
0b11 001
0b11 010
0b11 011
0b11 100
0b11 101
0b11 110
0b11 111

There are two lines with the same index

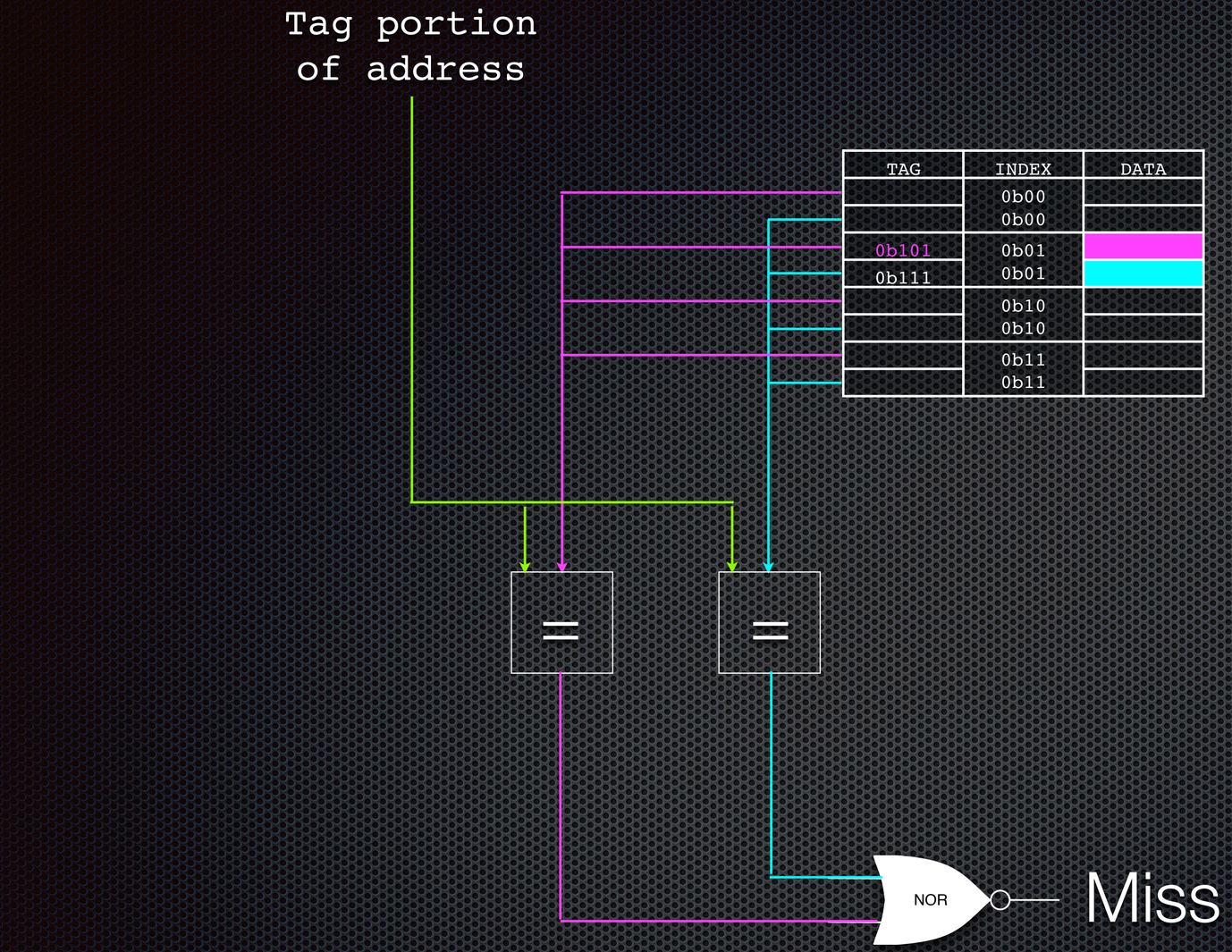


TAG	INDEX	DATA
	0b00	
	0b00	
0b101	0b01	
0b111	0b01	
	0b10	
	0b10	
	0b11	
	0b11	

Cache

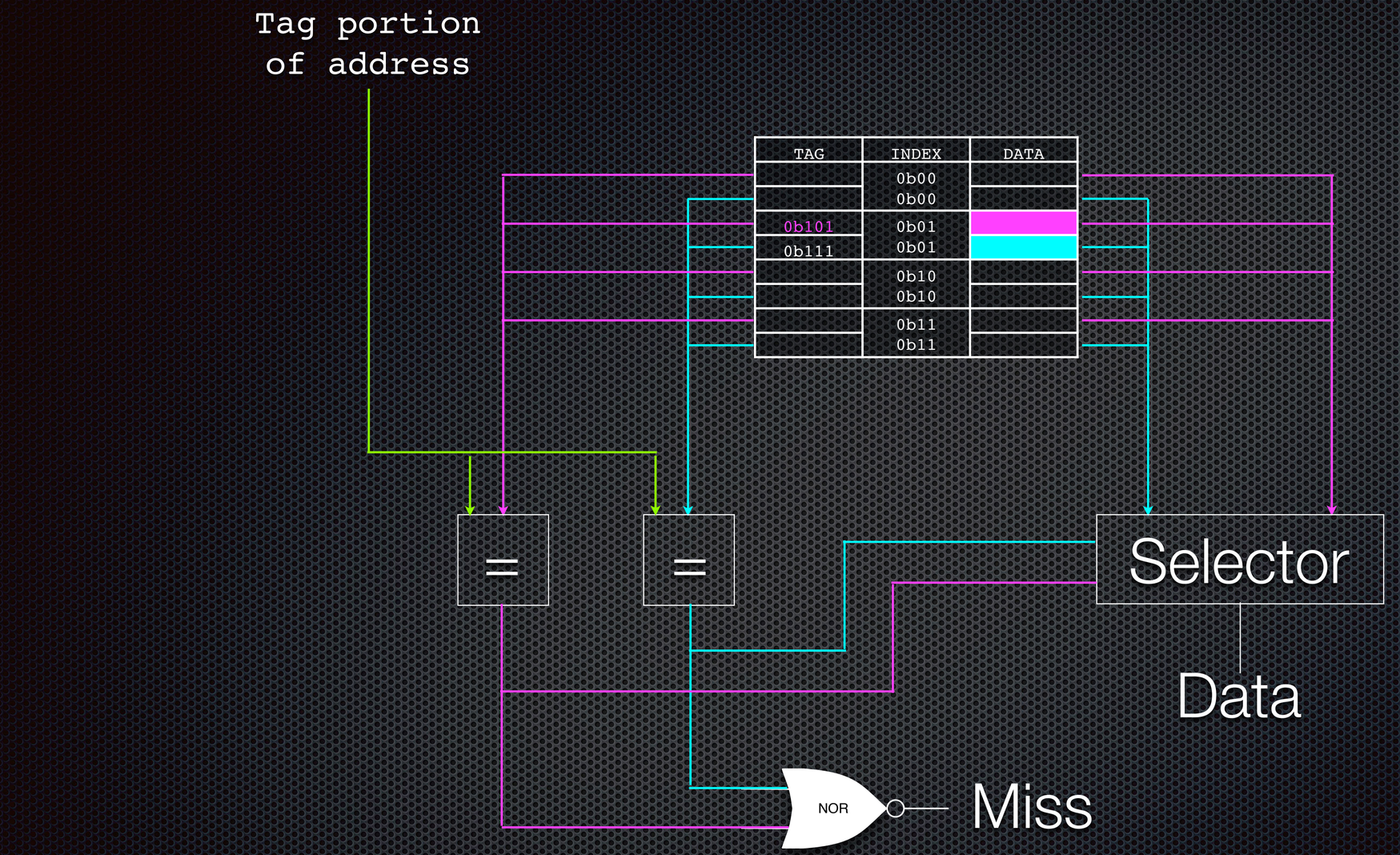
Reading

- When we read, the index segment of the address indicates where to look in cache
- But there are now two places where the target could be stored
- We have to compare the tag portion of the address with the tag in each of those lines


Tag portion
of address
TAG
INDEX
DATA
0b00
0b00
0b101
0b01
0b111
0b01
0b10
0b10
0b11
0b11
=
=
NOR
Miss

Data Access

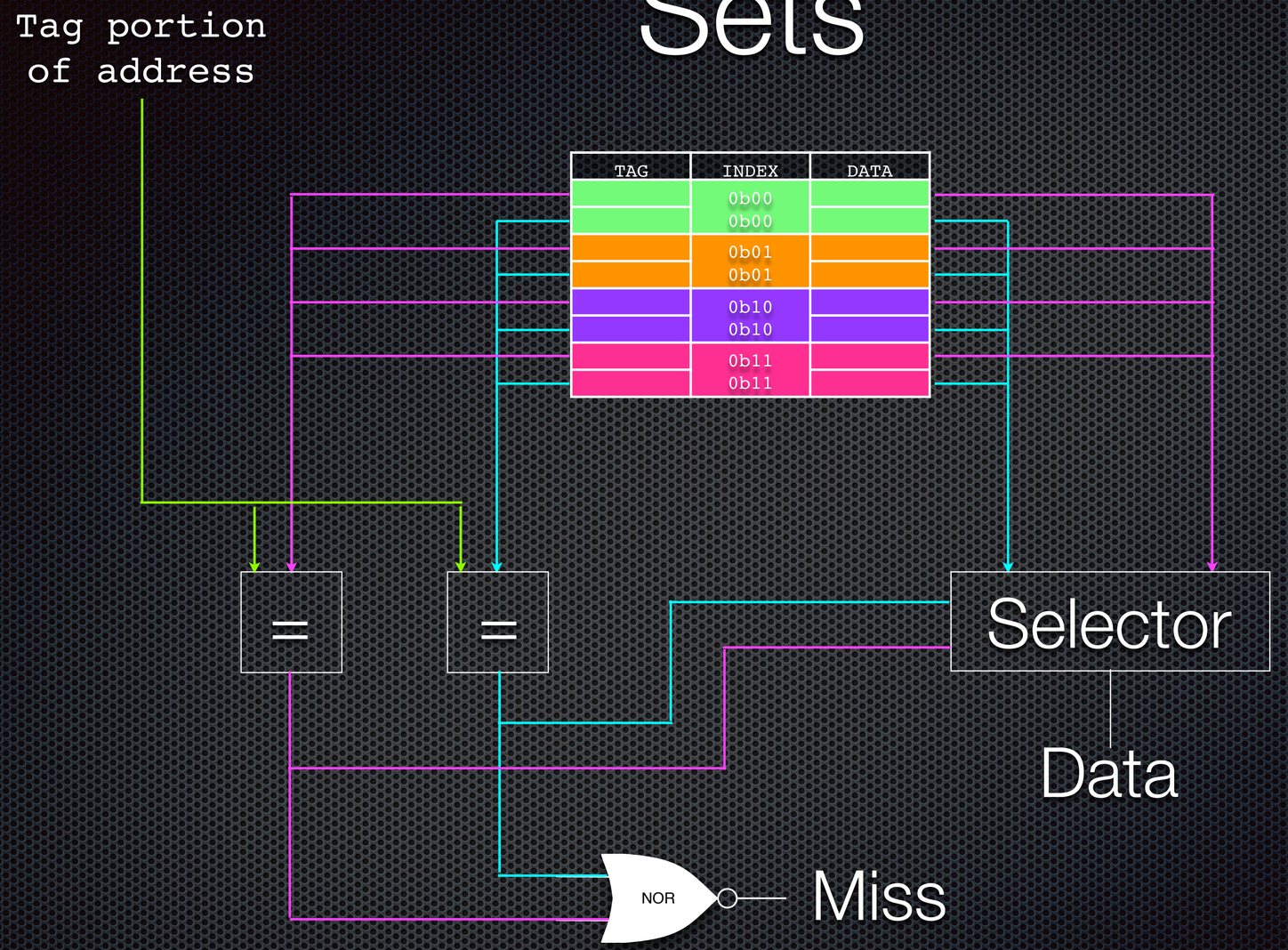
- If the read is a hit, we need to get the data from the matching line
- Can send a signal back to activate read logic on the line
- Faster to fetch both lines at same time as tag, and select one to output


Tag portion
of address
TAG
INDEX
DATA
0b00
0b00
0b101
0b01
0b111
0b01
0b10
0b10
0b11
0b11
=
=
Selector
Data
NOR
Miss

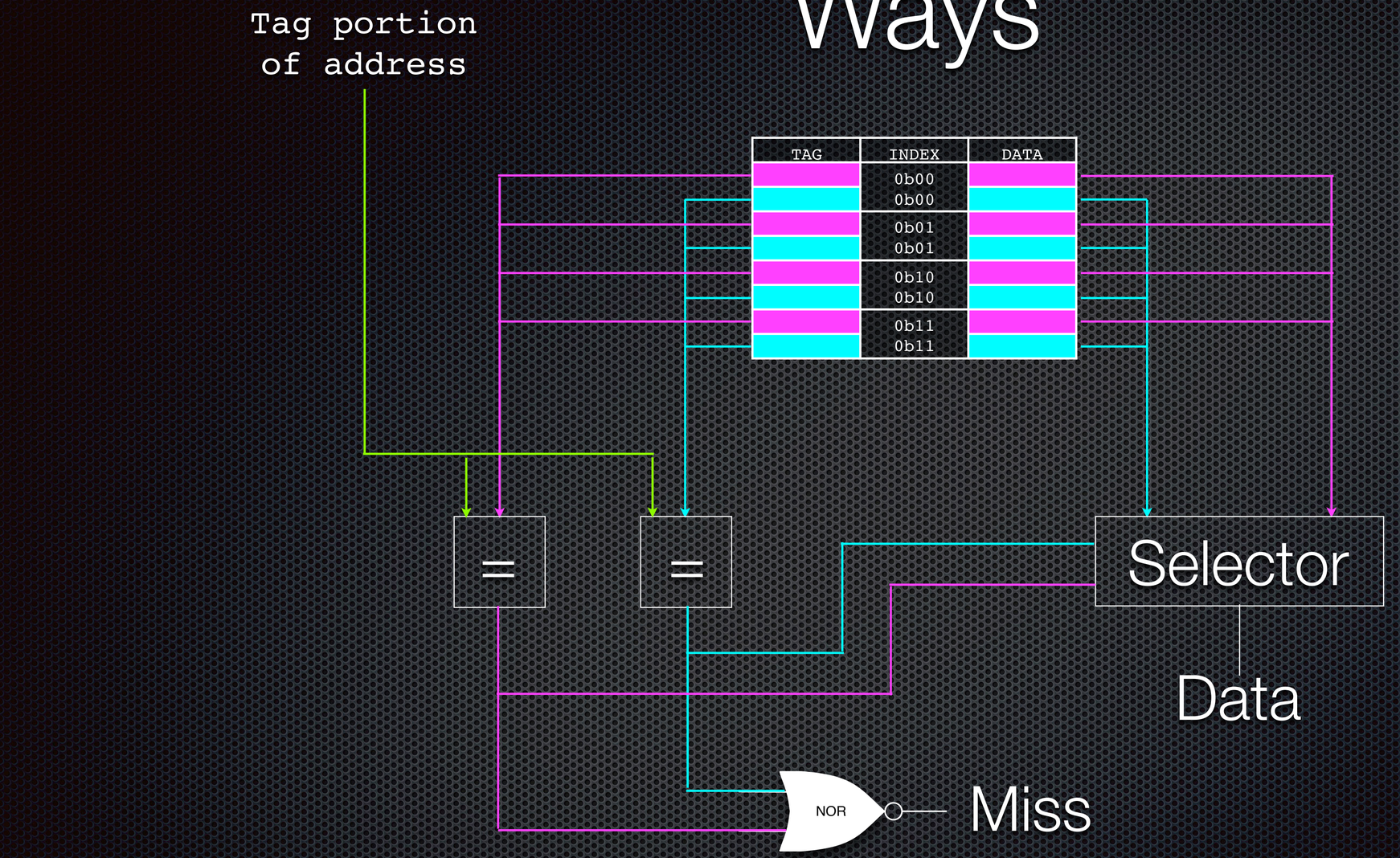
Associative Access

- Because we fetch and compare both tags at once, that portion of the addressing is by content (associative access)
- The group of 2 lines is called a set
- The collection of lines that are connected to each comparator is called a way

Sets


Tag portion
of address
TAG
INDEX
DATA
0b00
0b00
0b01
0b01
0b10
0b10
0b11
0b11
=
=
Selector
Data
NOR
Miss

Ways


Tag portion
of address
TAG
INDEX
DATA
0b00
0b00
0b01
0b01
0b10
0b10
0b11
0b11
=
=
Selector
Data
NOR
Miss

2-way Set Associative

- This cache has 2 ways
- It has #lines/#ways (8/2 = 4) sets
- Because each set contains one line connected to each way, we say it is 2-way set associative
- Associative access occurs within a set
- Addressed access occurs between sets

Higher Associativity

- If we increase the number of ways, the associativity increases
- Each set has more places to hold data from memory locations with conflicting indices
- Cache utilization and hit rate increase
- Number of comparators and buses increases

Fully Associative

- Every line has its own comparator
- No buses - a hit triggers read locally
- Maximum utilization and hit rate
- No index portion of address
- Access is only by tag content

Why Not Fully Associative?

- Every line has its own comparator and read select logic
  - Large logic overhead
  - Large power surges
  - Slow miss logic
- Not significantly better than 4- or 8-way set associative

Read Miss

- If the set is empty, load into the first line
- If the set is partly empty, load into the first available line
- If the set is full, something has to be evicted and replaced
- Direct mapped offers no replacement options
- Set associative is more complex

Eviction/Replacement

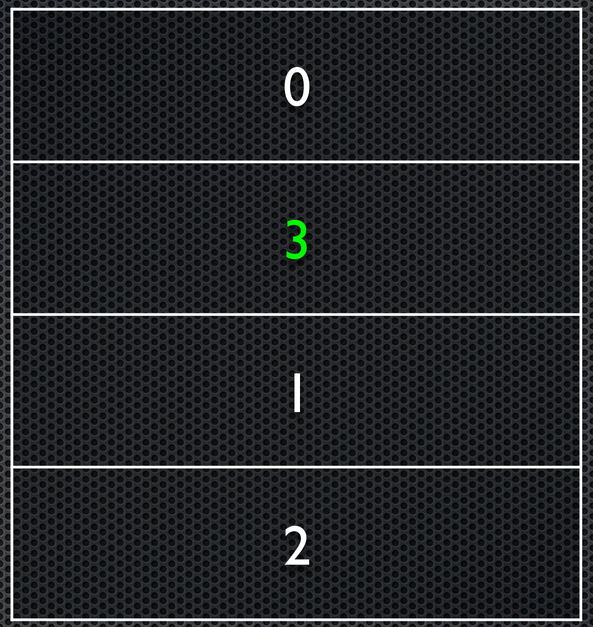
- FIFO / LRU / Random
- Allocate / No-allocate
- Write through / write back

LRU

- Simple for 2-way: Each access marks the other line as LRU
- For higher associativity, keep a number indicating use history
- On replacement, data goes to line with max LRU. History is set to 0, others incremented
- On non-replacement access, the accessed line is set to 0, and others less than it are incremented

Example: Replacement

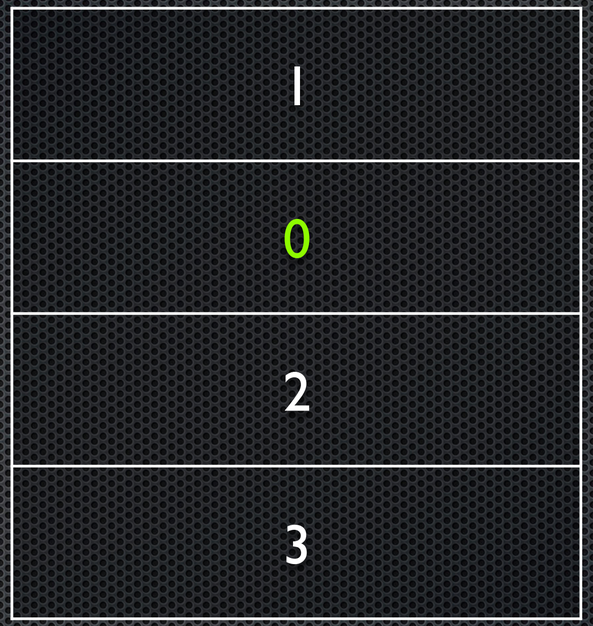
Line to evict:



In 4-way set associative, 3 = LRU

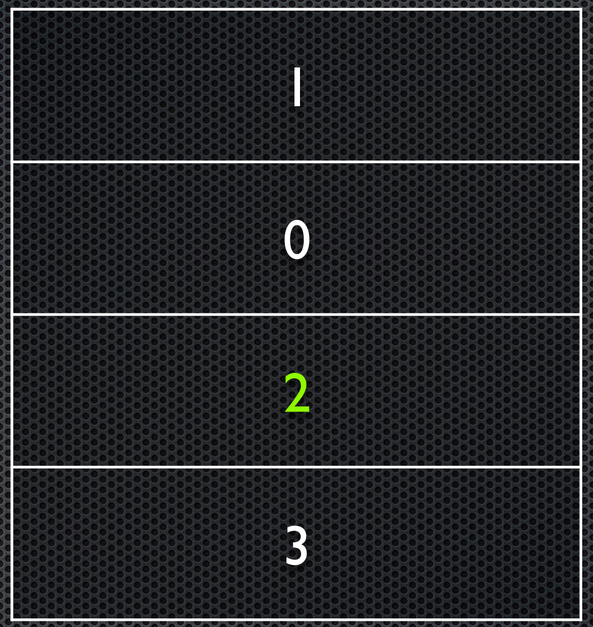
Example: Replacement

Replaced line:



Example: Access

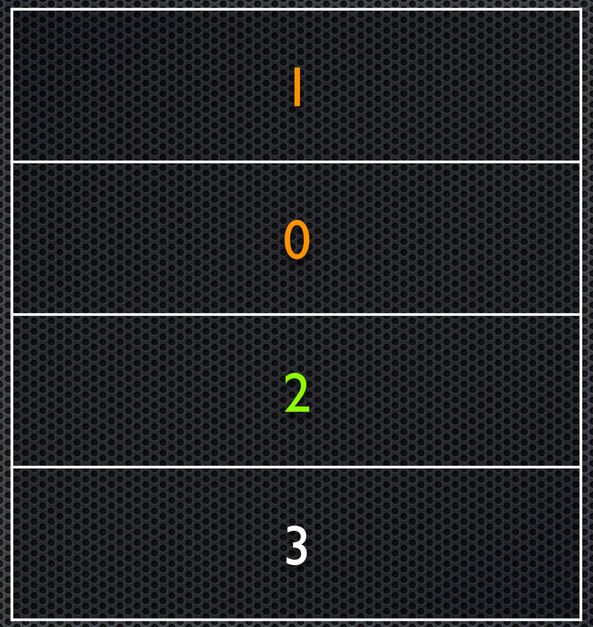
Accessed line:



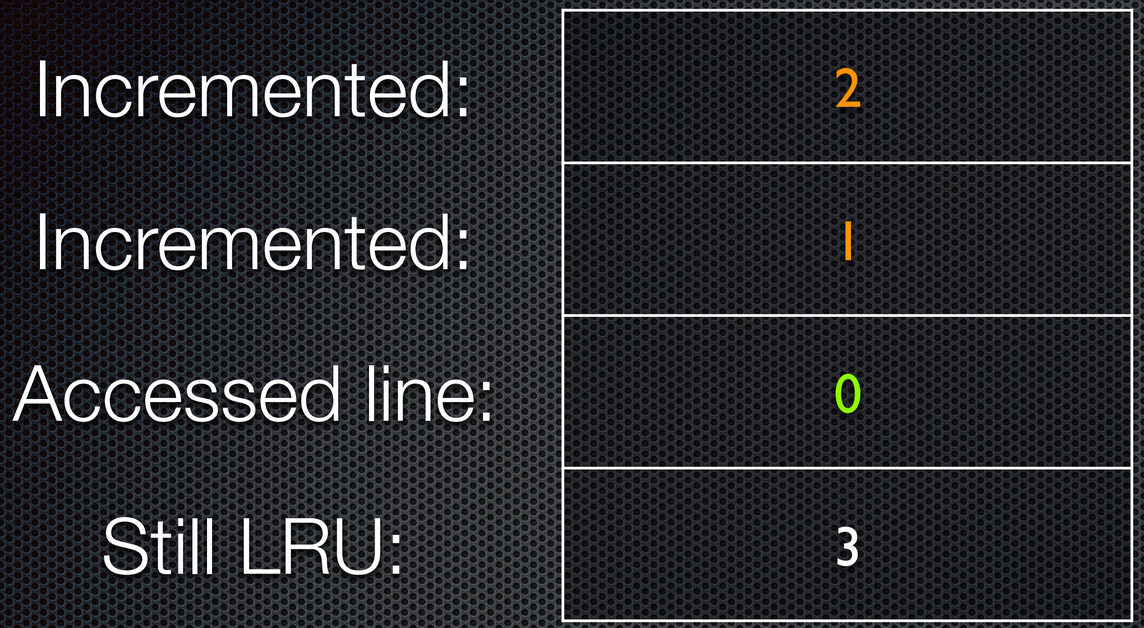
Example: Access

Lower value:

Lower value:



Example: Access



Replacement Issues

- FIFO and LRU have degenerate access patterns that can cause 0% hit rate
- Random is hard to implement, can have worse average hit rate, harder to reach peak rate
- Replacement policy must be fast, low cost
- Effect on hit rate decreases as cache size increases

Example

- 2-way set-associative cache
- 8 sets
- 2-word lines
- 40 memory references (perhaps out of 200 instructions), all reads for simplicity

References

00000000		10001110		01100101	
00000001		10001111		01110101	
00000010		00000000		01000101	
00000011		00000101		11100101	
00000100		00000011		01100101	
00000101		00000110		00000101	
00000110		10001000		00010101	
00000111		10001001		10000101	
10001000		00110001		11000111	
10001001		00110011		11001000	
10001010		00110110		11001001	
10001011		00110111		11101000	
10001100		00111000			
10001101		00101010			

Cache

Tag	Set #	Word 1	Word 0	LRU
	000			
	000			
	001			
	001			
	010			
	010			
	011			
	011			
	100			
	100			
	101			
	101			
	110			
	110			
	111			
	111			

Offset Doesn’t Matter to Cache

00000000		10001110		01100101	
00000001		10001111		01110101	
00000010		00000000		01000101	
00000011		00000101		11100101	
00000100		00000011		01100101	
00000101		00000110		00000101	
00000110		10001000		00010101	
00000111		10001001		10000101	
10001000		00110001		11000111	
10001001		00110011		11001000	
10001010		00110110		11001001	
10001011		00110111		11101000	
10001100		00111000			
10001101		00101010			

Tag	Set #	Word 1	Word 0	LRU
	000			
	000			
	001			
	001			
	010			
	010			
	011			
	011			
	100			
	100			
	101			
	101			
	110			
	110			
	111			
	111			

Which bit(s) can we ignore?

Low order bit is offset

00000000	10001110	01100101
00000001	10001111	01110101
00000010	00000000	01000101
00000011	00000101	11100101
00000100	00000011	01100101
00000101	00000110	00000101
00000110	10001000	00010101
00000111	10001001	10000101
10001000	00110001	11000111
10001001	00110011	11001000
10001010	00110110	11001001
10001011	00110111	11101000
10001100	00111000	
10001101	00101010	

Tag	Set #	Word 1	Word 0	LRU
	000			
	000			
	001			
	001			
	010			
	010			
	011			
	011			
	100			
	100			
	101			
	101			
	110			
	110			
	111			
	111			

How many bits are in the index?

00000000	10001110	01100101
00000001	10001111	01110101
00000010	00000000	01000101
00000011	00000101	11100101
00000100	00000011	01100101
00000101	00000110	00000101
00000110	10001000	00010101
00000111	10001001	10000101
10001000	00110001	11000111
10001001	00110011	11001000
10001010	00110110	11001001
10001011	00110111	11101000
10001100	00111000	
10001101	00101010	

Tag	Set #	Word 1	Word 0	LRU
	000			
	000			
	001			
	001			
	010			
	010			
	011			
	011			
	100			
	100			
	101			
	101			
	110			
	110			
	111			
	111			

How many bits are in the index?

00000000	10001110	01100101
00000001	10001111	01110101
00000010	00000000	01000101
00000011	00000101	11100101
00000100	00000011	01100101
00000101	00000110	00000101
00000110	10001000	00010101
00000111	10001001	10000101
10001000	00110001	11000111
10001001	00110011	11001000
10001010	00110110	11001001
10001011	00110111	11101000
10001100	00111000	
10001101	00101010	

Tag	Set #	Word 1	Word 0	LRU
	000			
	000			
	001			
	001			
	010			
	010			
	011			
	011			
	100			
	100			
	101			
	101			
	110			
	110			
	111			
	111			

With 8 sets, it takes 3 bits to index them

Upper 4 bits are thus the tag

00000000	10001110	01100101
00000001	10001111	01110101
00000010	00000000	01000101
00000011	00000101	11100101
00000100	00000011	01100101
00000101	00000110	00000101
00000110	10001000	00010101
00000111	10001001	10000101
10001000	00110001	11000111
10001001	00110011	11001000
10001010	00110110	11001001
10001011	00110111	11101000
10001100	00111000	
10001101	00101010	

Tag	Set #	Word 1	Word 0	LRU
	000			
	000			
	001			
	001			
	010			
	010			
	011			
	011			
	100			
	100			
	101			
	101			
	110			
	110			
	111			
	111			

We’re ready to start the trace

Index is 000, tag is 0000: Miss

00000000	M	10001110		01100101	
00000001		10001111		01110101	
00000010		00000000		01000101	
00000011		00000101		11100101	
00000100		00000011		01100101	
00000101		00000110		00000101	
00000110		10001000		00010101	
00000111		10001001		10000101	
10001000		00110001		11000111	
10001001		00110011		11001000	
10001010		00110110		11001001	
10001011		00110111		11101000	
10001100		00111000			
10001101		00101010			

Tag	Set #	Word 1	Word 0	LRU
0000	000			0
	000			1
	001			
	001			
	010			
	010			
	011			
	011			
	100			
	100			
	101			
	101			
	110			
	110			
	111			
	111			

Set LRU to 0, both words are loaded

Index is 000, tag is 0000: Hit

00000000	M	10001110		01100101	
00000001	H	10001111		01110101	
00000010		00000000		01000101	
00000011		00000101		11100101	
00000100		00000011		01100101	
00000101		00000110		00000101	
00000110		10001000		00010101	
00000111		10001001		10000101	
10001000		00110001		11000111	
10001001		00110011		11001000	
10001010		00110110		11001001	
10001011		00110111		11101000	
10001100		00111000					
10001101		00101010					

Tag	Set #	Word 1	Word 0	LRU
0000	000			0
	000			1
	001			
	001			
	010			
	010			
	011			
	011			
	100			
	100			
	101			
	101			
	110			
	110			
	111			
	111			

LRU remains 0

Index is 001, tag is 0000: Miss

00000000		M		10001110		01100101	
00000001		H		10001111		01110101	
00000010		M		00000000		01000101	
00000011				00000101		11100101	
00000100				00000011		01100101	
00000101				00000110		00000101	
00000110				10001000		00010101	
00000111				10001001		10000101	
10001000				00110001		11000111	
10001001				00110011		11001000	
10001010				00110110		11001001	
10001011				00110111		11101000	
10001100				00111000			
10001101				00101010			

Tag	Set #	Word 1	Word 0	LRU
0000	000			0
	000			1
0000	001			0
	001			1
	010			
	010			
	011			
	011			
	100			
	100			
	101			
	101			
	110			
	110			
	111			
	111			

Set LRU to 0, both words are loaded

Index is 001, tag is 0000: Hit

00000000	M	10001110		01100101	
00000001	H	10001111		01110101	
00000010	M	00000000		01000101	
00000011	H	00000101		11100101	
00000100		00000011		01100101	
00000101		00000110		00000101	
00000110		10001000		00010101	
00000111		10001001		10000101	
10001000		00110001		11000111	
10001001		00110011		11001000	
10001010		00110110		11001001	
10001011		00110111		11101000	
10001100		00111000			
10001101		00101010			

Tag	Set #	Word 1	Word 0	LRU
0000	000			0
	000			1
0000	001			0
	001			1
	010			
	010			
	011			
	011			
	100			
	100			
	101			
	101			
	110			
	110			
	111			
	111			

LRU remains 0

Fast Forward

Index is 111, tag is 1000: Hit

00000000	M	10001110	M	01100101	
00000001	H	10001111	H	01110101	
00000010	M	00000000		01000101	
00000011	H	00000101		11100101	
00000100	M	00000011		01100101	
00000101	H	00000110		00000101	
00000110	M	10001000		00010101	
00000111	H	10001001		10000101	
10001000	M	00110001		11000111	
10001001	H	00110011		11001000	
10001010	M	00110110		11001001	
10001011	H	00110111		11101000	
10001100	M	00111000			
10001101	H	00101010			

Tag	Set #	Word 1	Word 0	LRU
0000	000			0
	000			1
0000	001			0
	001			1
0000	010			0
	010			1
0000	011			0
	011			1
1000	100			0
	100			1
1000	101			0
	101			1
1000	110			0
	110			1
1000	111			0
	111			1

LRU remains 0

There then follows a series of hits

00000000	M	10001110	M	01100101			
00000001	H	10001111	H	01110101			
00000010	M	00000000	H	01000101			
00000011	H	00000101	H	11100101			
00000100	M	00000011	H	01100101			
00000101	H	00000110	H	00000101			
00000110	M	10001000	H	00010101			
00000111	H	10001001	H	10000101			
10001000	M	00110001		11000111			
10001001	H	00110011		11001000			
10001010	M	00110110		11001001			
10001011	H	00110111		11101000			
10001100	M	00111000					
10001101	H	00101010					

Tag	Set #	Word 1	Word 0	LRU
0000	000			0
	000			1
0000	001			0
	001			1
0000	010			0
	010			1
0000	011			0
	011			1
1000	100			0
	100			1
1000	101			0
	101			1
1000	110			0
	110			1
1000	111			0
	111			1

And at last it gets interesting

Index 000, Tag 0011: Miss (would evict in DMC)

00000000	M	10001110	M	01100101	
00000001	H	10001111	H	01110101	
00000010	M	00000000	H	01000101	
00000011	H	00000101	H	11100101	
00000100	M	00000011	H	01100101	
00000101	H	00000110	H	00000101	
00000110	M	10001000	H	00010101	
00000111	H	10001001	H	10000101	
10001000	M	00110001	M	11000111	
10001001	H	00110011		11001000	
10001010	M	00110110		11001001	
10001011	H	00110111		11101000	
10001100	M	00111000			
10001101	H	00101010			

Tag	Set #	Word 1	Word 0	LRU
0000	000			1
0011	000			0
0000	001			0
	001			1
0000	010			0
	010			1
0000	011			0
	011			1
1000	100			0
	100			1
1000	101			0
	101			1
1000	110			0
	110			1
1000	111			0
	111			1

Fills second way of set 0 -- becomes MRU

Index 000, Tag 0011: Hit

00000000	M	10001110	M	01100101			
00000001	H	10001111	H	01110101			
00000010	M	00000000	H	01000101			
00000011	H	00000101	H	11100101			
00000100	M	00000011	H	01100101			
00000101	H	00000110	H	00000101			
00000110	M	10001000	H	00010101			
00000111	H	10001001	H	10000101			
10001000	M	00110001	M	11000111			
10001001	H	00110011	M	11001000			
10001010	M	00110110		11001001			
10001011	H	00110111		11101000			
10001100	M	00111000					
10001101	H	00101010					

Tag	Set #	Word 1	Word 0	LRU
0000	000			1
0011	000			0
0000	001			1
0011	001			0
0000	010			0
	010			1
0000	011			0
0011	011			1
1000	100			0
	100			1
1000	101			0
	101			1
1000	110			0
	110			1
1000	111			0
	111			1

Remains MRU

Another miss and subsequent hit

00000000	M	10001110	M	01100101			
00000001	H	10001111	H	01110101			
00000010	M	00000000	H	01000101			
00000011	H	00000101	H	11100101			
00000100	M	00000011	H	01100101			
00000101	H	00000110	H	00000101			
00000110	M	10001000	H	00010101			
00000111	H	10001001	H	10000101			
10001000	M	00110001	M	11000111			
10001001	H	00110011	M	11001000			
10001010	M	00110110	M	11001001			
10001011	H	00110111	H	11101000			
10001100	M	00111000					
10001101	H	00101010					

Tag	Set #	Word 1	Word 0	LRU
0000	000			1
0011	000			0
0000	001			1
0011	001			0
0000	010			0
	010			1
0000	011			1
0011	011			0
1000	100			0
	100			1
1000	101			0
	101			1
1000	110			0
	110			1
1000	111			0
	111			1

Spatial locality in action

3 more misses fill most of way 1

00000000	M	10001110	M	01100101	M
00000001	H	10001111	H	01110101	
00000010	M	00000000	H	01000101	
00000011	H	00000101	H	11100101	
00000100	M	00000011	H	01100101	
00000101	H	00000110	H	00000101	
00000110	M	10001000	H	00010101	
00000111	H	10001001	H	10000101	
10001000	M	00110001	M	11000111	
10001001	H	00110011	M	11001000	
10001010	M	00110110	M	11001001	
10001011	H	00110111	H	11101000	
10001100	M	00111000	M		
10001101	H	00101010	M		

Tag	Set #	Word 1	Word 0	LRU
0000	000			1
0011	000			0
0000	001			1
0011	001			0
0000	010			1
0110	010			0
0000	011			1
0011	011			0
1000	100			1
0011	100			0
1000	101			1
0010	101			0
1000	110			1
	110			0
1000	111			0
	111			1

Next we have an eviction

Index 010, Tag 0111: Miss

00000000	M	10001110	M	01100101	M		
00000001	H	10001111	H	01110101	M		
00000010	M	00000000	H	01000101			
00000011	H	00000101	H	11100101			
00000100	M	00000011	H	01100101			
00000101	H	00000110	H	00000101			
00000110	M	10001000	H	00010101			
00000111	H	10001001	H	10000101			
10001000	M	00110001	M	11000111			
10001001	H	00110011	M	11001000			
10001010	M	00110110	M	11001001			
10001011	H	00110111	H	11101000			
10001100	M	00111000	M				
10001101	H	00101010	M				

Tag	Set #	Word 1	Word 0	LRU
0000	000			1
0011	000			0
0000	001			1
0011	001			0
0000	010			1
0110	010			0
0000	011			1
0011	011			0
1000	100			1
0011	100			0
1000	101			1
0010	101			0
1000	110			1
	110			0
1000	111			0
	111			1

Select way 0 (LRU) for eviction

Change tag and LRU status

00000000	M	10001110	M	01100101	M		
00000001	H	10001111	H	01110101	M		
00000010	M	00000000	H	01000101			
00000011	H	00000101	H	11100101			
00000100	M	00000011	H	01100101			
00000101	H	00000110	H	00000101			
00000110	M	10001000	H	00010101			
00000111	H	10001001	H	10000101			
10001000	M	00110001	M	11000111			
10001001	H	00110011	M	11001000			
10001010	M	00110110	M	11001001			
10001011	H	00110111	H	11101000			
10001100	M	00111000	M				
10001101	H	00101010	M				

Tag	Set #	Word 1	Word 0	LRU
0000	000			1
0011	000			0
0000	001			1
0011	001			0
0111	010			0
0110	010			1
0000	011			1
0011	011			0
1000	100			1
0011	100			0
1000	101			1
0010	101			0
1000	110			1
	110			0
1000	111			0
	111			1

Load new data into both words

Index 010, Tag 0100: Miss

00000000	M	10001110	M	01100101	M		
00000001	H	10001111	H	01110101	M		
00000010	M	00000000	H	01000101	M		
00000011	H	00000101	H	11100101			
00000100	M	00000011	H	01100101			
00000101	H	00000110	H	00000101			
00000110	M	10001000	H	00010101			
00000111	H	10001001	H	10000101			
10001000	M	00110001	M	11000111			
10001001	H	00110011	M	11001000			
10001010	M	00110110	M	11001001			
10001011	H	00110111	H	11101000			
10001100	M	00111000	M				
10001101	H	00101010	M				

Tag	Set #	Word 1	Word 0	LRU
0000	000			1
0011	000			0
0000	001			1
0011	001			0
0111	010			0
0110	010			1
0000	011			1
0011	011			0
1000	100			1
0011	100			0
1000	101			1
0010	101			0
1000	110			1
	110			0
1000	111			0
	111			1

Replace LRU line

Update tag and LRU, fetch data

00000000	M	10001110	M	01100101	M		
00000001	H	10001111	H	01110101	M		
00000010	M	00000000	H	01000101	M		
00000011	H	00000101	H	11100101			
00000100	M	00000011	H	01100101			
00000101	H	00000110	H	00000101			
00000110	M	10001000	H	00010101			
00000111	H	10001001	H	10000101			
10001000	M	00110001	M	11000111			
10001001	H	00110011	M	11001000			
10001010	M	00110110	M	11001001			
10001011	H	00110111	H	11101000			
10001100	M	00111000	M				
10001101	H	00101010	M				

Tag	Set #	Word 1	Word 0	LRU
0000	000			1
0011	000			0
0000	001			1
0011	001			0
0111	010			1
0100	010			0
0000	011			1
0011	011			0
1000	100			1
0011	100			0
1000	101			1
0010	101			0
1000	110			1
	110			0
1000	111			0
	111			1

Another eviction in 010

00000000	M	10001110	M	01100101	M
00000001	H	10001111	H	01110101	M
00000010	M	00000000	H	01000101	M
00000011	H	00000101	H	11100101	M
00000100	M	00000011	H	01100101	
00000101	H	00000110	H	00000101	
00000110	M	10001000	H	00010101	
00000111	H	10001001	H	10000101	
10001000	M	00110001	M	11000111	
10001001	H	00110011	M	11001000	
10001010	M	00110110	M	11001001	
10001011	H	00110111	H	11101000	
10001100	M	00111000	M		
10001101	H	00101010	M		

Tag	Set #	Word 1	Word 0	LRU
0000	000			1
0011	000			0
0000	001			1
0011	001			0
1110	010			0
0100	010			1
0000	011			1
0011	011			0
1000	100			1
0011	100			0
1000	101			1
0010	101			0
1000	110			1
	110			0
1000	111			0
	111			1

And another

00000000	M	10001110	M	01100101	M
00000001	H	10001111	H	01110101	M
00000010	M	00000000	H	01000101	M
00000011	H	00000101	H	11100101	M
00000100	M	00000011	H	01100101	M
00000101	H	00000110	H	00000101	
00000110	M	10001000	H	00010101	
00000111	H	10001001	H	10000101	
10001000	M	00110001	M	11000111	
10001001	H	00110011	M	11001000	
10001010	M	00110110	M	11001001	
10001011	H	00110111	H	11101000	
10001100	M	00111000	M		
10001101	H	00101010	M		

Tag	Set #	Word 1	Word 0	LRU
0000	000			1
0011	000			0
0000	001			1
0011	001			0
1110	010			1
0110	010			0
0000	011			1
0011	011			0
1000	100			1
0011	100			0
1000	101			1
0010	101			0
1000	110			1
	110			0
1000	111			0
	111			1

But this one has been in cache before

00000000	M	10001110	M	01100101	M		
00000001	H	10001111	H	01110101	M		
00000010	M	00000000	H	01000101	M		
00000011	H	00000101	H	11100101	M		
00000100	M	00000011	H	01100101	M		
00000101	H	00000110	H	00000101			
00000110	M	10001000	H	00010101			
00000111	H	10001001	H	10000101			
10001000	M	00110001	M	11000111			
10001001	H	00110011	M	11001000			
10001010	M	00110110	M	11001001			
10001011	H	00110111	H	11101000			
10001100	M	00111000	M				
10001101	H	00101010	M				

Tag	Set #	Word 1	Word 0	LRU
0000	000			1
0011	000			0
0000	001			1
0011	001			0
1110	010			1
0110	010			0
0000	011			1
0011	011			0
1000	100			1
0011	100			0
1000	101			1
0010	101			0
1000	110			1
	110			0
1000	111			0
	111			1

A conflict miss

This one has also been here previously

00000000	M	10001110	M	01100101	M		
00000001	H	10001111	H	01110101	M		
00000010	M	00000000	H	01000101	M		
00000011	H	00000101	H	11100101	M		
00000100	M	00000011	H	01100101	M		
00000101	H	00000110	H	00000101	M		
00000110	M	10001000	H	00010101			
00000111	H	10001001	H	10000101			
10001000	M	00110001	M	11000111			
10001001	H	00110011	M	11001000			
10001010	M	00110110	M	11001001			
10001011	H	00110111	H	11101000			
10001100	M	00111000	M				
10001101	H	00101010	M				

Tag	Set #	Word 1	Word 0	LRU
0000	000			1
0011	000			0
0000	001			1
0011	001			0
0000	010			0
0110	010			1
0000	011			1
0011	011			0
1000	100			1
0011	100			0
1000	101			1
0010	101			0
1000	110			1
	110			0
1000	111			0
	111			1

A second conflict miss

Two more compulsory misses

00000000	M	10001110	M	01100101	M		
00000001	H	10001111	H	01110101	M		
00000010	M	00000000	H	01000101	M		
00000011	H	00000101	H	11100101	M		
00000100	M	00000011	H	01100101	M		
00000101	H	00000110	H	00000101	M		
00000110	M	10001000	H	00010101	M		
00000111	H	10001001	H	10000101	M		
10001000	M	00110001	M	11000111			
10001001	H	00110011	M	11001000			
10001010	M	00110110	M	11001001			
10001011	H	00110111	H	11101000			
10001100	M	00111000	M				
10001101	H	00101010	M				

Tag	Set #	Word 1	Word 0	LRU
0000	000			1
0011	000			0
0000	001			1
0011	001			0
1000	010			0
0001	010			1
0000	011			1
0011	011			0
1000	100			1
0011	100			0
1000	101			1
0010	101			0
1000	110			1
	110			0
1000	111			0
	111			1

In the end

00000000	M	10001110	M	01100101	M
00000001	H	10001111	H	01110101	M
00000010	M	00000000	H	01000101	M
00000011	H	00000101	H	11100101	M
00000100	M	00000011	H	01100101	M
00000101	H	00000110	H	00000101	M
00000110	M	10001000	H	00010101	M
00000111	H	10001001	H	10000101	M
10001000	M	00110001	M	11000111	M
10001001	H	00110011	M	11001000	M
10001010	M	00110110	M	11001001	H
10001011	H	00110111	H	11101000	M
10001100	M	00111000	M		
10001101	H	00101010	M		

Tag	Set #	Word 1	Word 0	LRU
0000	000			1
0011	000			0
0000	001			1
0011	001			0
1000	010			0
0001	010			1
1100	011			0
0011	011			1
1100	100			1
1110	100			0
1000	101			1
0010	101			0
1000	110			1
	110			0
1000	111			0
	111			1

16/40 = 40% hit rate

Notes

- Cache never warmed up -- still empty lines
- Trace is too small for significance
- Need at least 100 misses per line after warmup to have statistical significance
- Simulation can be very long

Types of Misses

- Compulsory -- never in cache before
- Conflict -- was in cache before, and evicted by a value mapped to the same index
- Capacity -- was evicted due to lack of space

Cache Add-Ons

- Write Buffer: holds writes from CPU until cache can store
- MSHR -- coalesces multiple misses to same line
- Hot Word First -- get the requested word before others in the line
- Victim Buffer -- saves recently evicted lines
- Split I/D -- allows separate configurations
- Prefetch -- read ahead from memory
- Stream Buffer -- catches streaming data

Multi-Level Caches

Pyramid scheme for data

Why Levels?

- As cache size increases, wires get longer
- Longer wires have more resistance and capacitance
- Delay is proportional to RC product
- L1 cache should respond in one or few cycles, so when delay exceeds time, all accesses slow down
- Limit size to meet time goal, then add a level

Typical Cache Parameters

- L1D 32KB, 8-way, 64-byte line, 4 cycle (pipelined)
- L1I 32KB, 8-way, 64-byte line, 4 cycle (pipelined)
- L2 Unified, 256KB, 64-byte line, 11 cycle
- L3 Unified, 8MB, 64-byte line, 38 cycle
- DRAM, GB, 64-byte line, 100 cycle
- Note common line lengths

Extending the Basic Model

Miss buffer, Victim buffer, Stream buffer

Write Buffer

- Writes can be to non-cached locations
- With superscalar pipes, cache can be busy on write
- Blocked writes prevent reads (dependences)
- Provide a small queue to hold pending writes
- Fully associative for checking subsequent reads
- Improves performance but makes consistency and coherence tricky to implement for shared memory

Non-blocking

- During a miss, cache is busy
- Accesses from superscalar pipes will block and stall
- Add support to access independent lines during miss
- Add queue for additional misses

MSHR

- Misses from superscalar pipes can be clustered
  - Miss from one pipe starts being serviced
  - Miss from another pipe is to the same line
  - Fills queue entries in non-blocking cache
- Miss Status Handling Registers catch new misses to lines currently pending, and coalesce for service

Hot Word First

- Misses are serviced with accesses to lower level cache or RAM
- Simplest scheme is to fill the cache line and then complete the read cycle
- Additional logic can extract the requested word as it comes in from the lower level, and forward it directly to the processor

Trace Cache

- Intel Netburst
- Specialized instruction cache
- Records fully decoded ops as they exit completion unit
- Trace ends with condition such as indirect branch
- Detecting entry into a trace, the issue engine substitutes the trace entries while checking that the actual instructions don’t violate the prediction
- Performance can be brittle (subject to minor changes)

DASAT

- Combines a short, fully associative, wide, spatial buffer with two long, narrow, direct mapped, temporal caches
- Spatial buffer keeps stats on accesses
- Evicted hot words from spatial buffer promoted to cache, cold words return to L2, evictions from cache return to L2
- Spatial buffer notes access runs and dynamically prefetches to keep ahead
- Permits small L1 cache to match a cache twice its size

Cache Optimizations

- More L1 associativity
- Multibanked caches
- Way/block/bank prediction
- Pipelined cache
- Nonblocking caches (10 - 15%)
- Critical word first
- Merging write buffer
- Miss status handling registers
- Compiler prefetching

Compiler Optimizations

- Loop interchange
- Blocking
- Unrolling
- Loop fusion
- Global code motion

Statistically-Based Cache Enhancements

- Increased size, longer lines, associativity, miss coalescing (MSHRs), and victim buffers reduce or accelerate misses
- These are all based on statistical properties of general software behavior, rather than specific actions
- As a result, they are relatively robust, simple to implement, and reasonably effective

How Effective is Enough?

- From the memory wall paper, we saw that even perfect caching isn’t enough
- As higher clock rate and instruction level parallelism (superscalar pipelines, etc.) increase instruction rate, and memory latency lags, even a small miss rate can be a limiting factor
- Eliminating capacity and conflict misses would still leave compulsory misses

Software Specific Cache Enhancements

- Use more specific software behavior to avoid misses (even “compulsory” ones)
- Stream buffers target unit-stride access patterns
- Prefetch instructions allow compiler analyses to signal impending loads and precompute addresses
- Evict-me-next loads enable tagging locations on last reference for early eviction

Compilers aren’t Omniscient

- Some analyses depend on data that is only available at run time
- Profiling provides general run time guidance, but can’t capture specific cases
- Run time isn’t omniscient either -- context is local, and lacks compiler’s broad structural information

Sim ISCA 2013

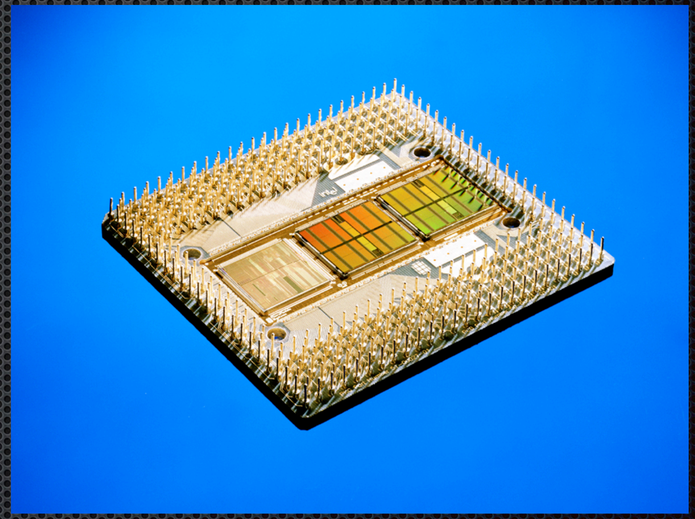
Resilient Die-Stacked DRAM Caches

Problem

- DRAM isn’t shrinking
- DRAM isn’t getting faster
- DRAM is error prone
- Increasing core count needs
  - More memory
  - Higher memory bandwidth
  - Low error rates

Solution?

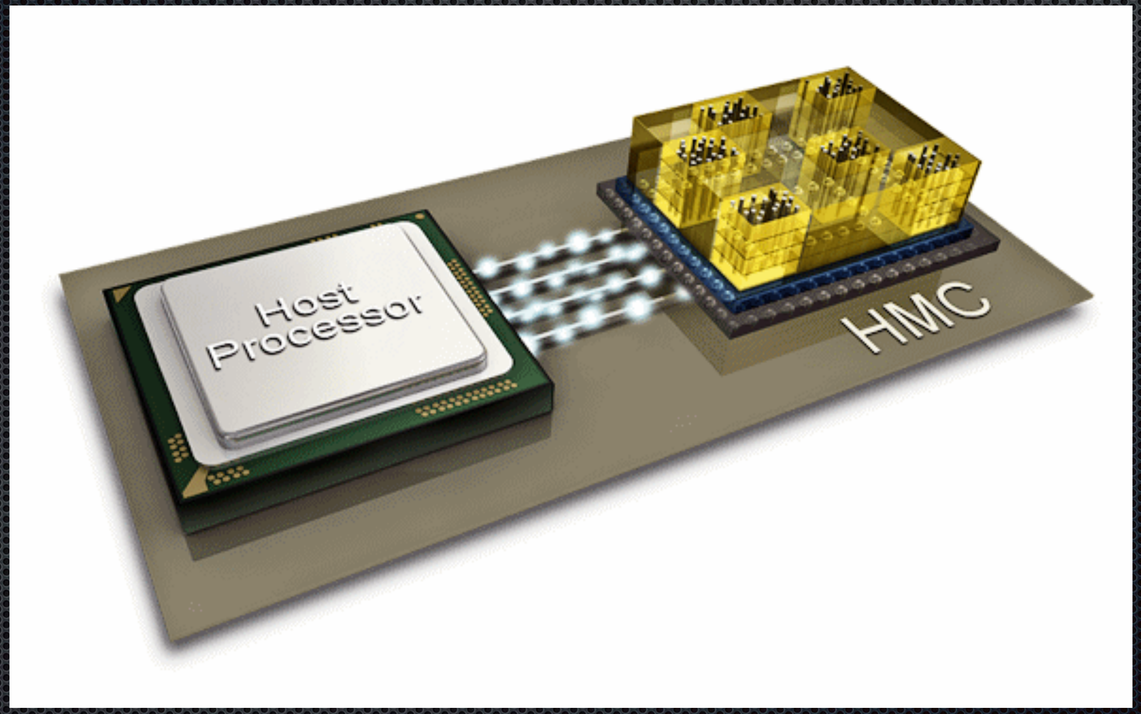
- Take DRAM chips out their separate packages
- Stack them directly on top of the processor
  - Reduces latency (shorter wires)
  - Increases bandwidth (more pins)
  - More expensive, not repairable
  - Limited fast memory (L4 cache)
- Pentium Pro for the 2020's?



Pentium Pro (1995)

Commercial Concept

Micron Hybrid Memory Cube

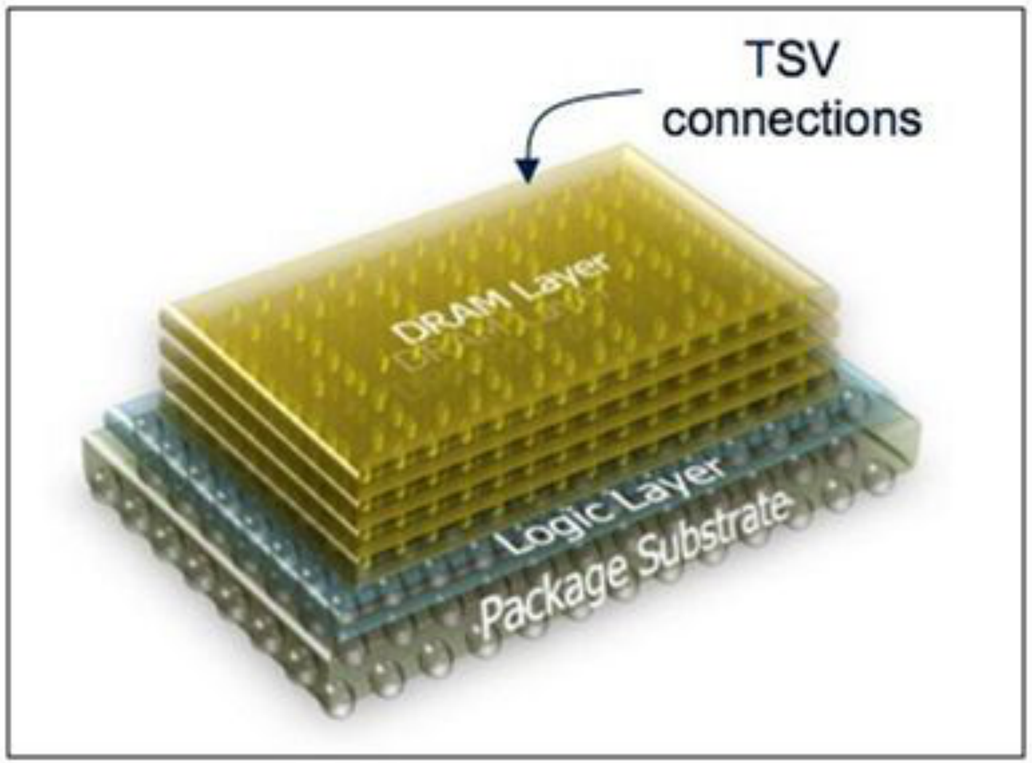


DRAM stacks in separate module

HMC Module

Through Silicon Vias (TSV)

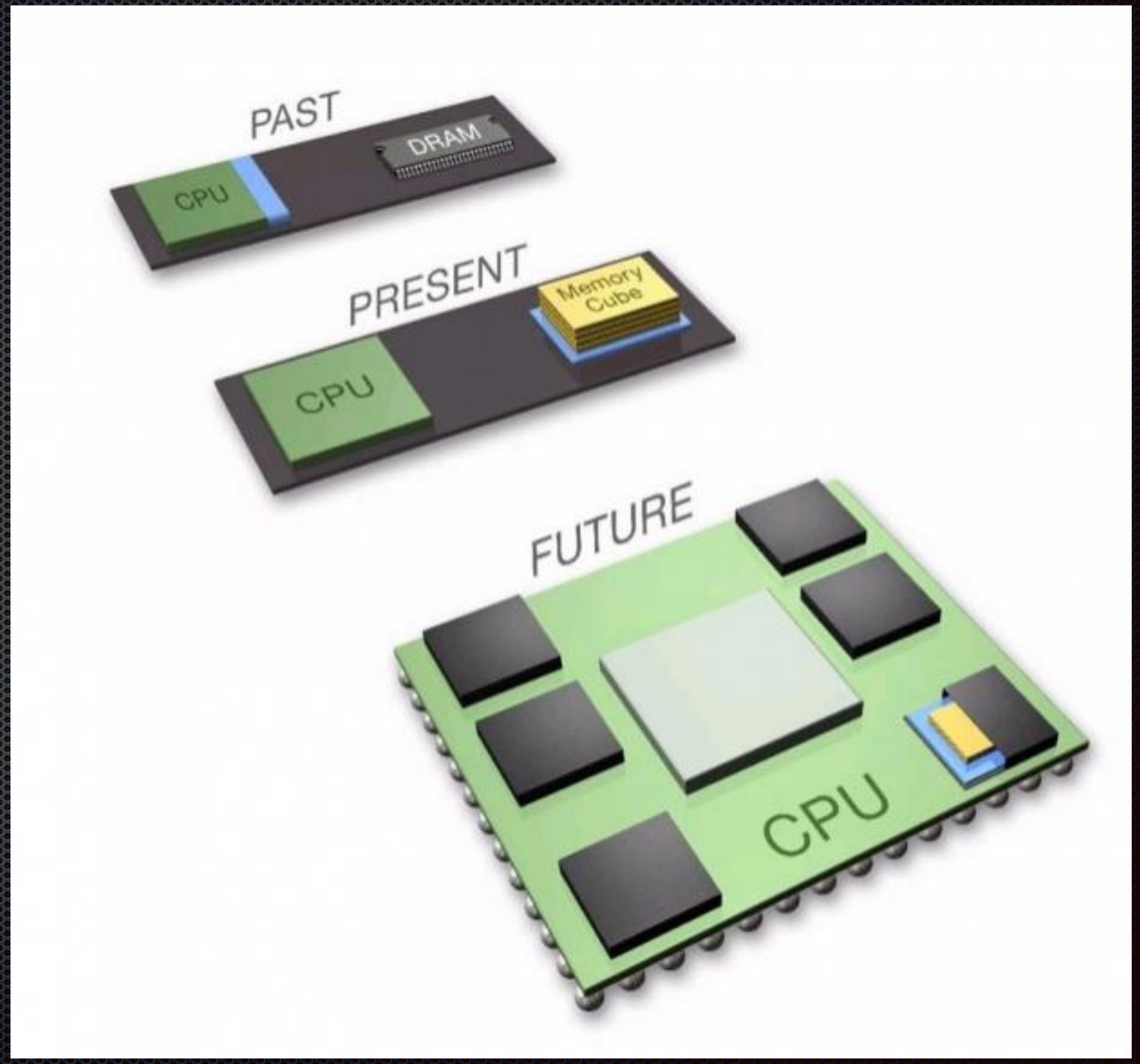
DRAM stack on top of
logic chip that provides
interface (crossbar switch)





Looking Ahead

This paper explores the characteristics needed to integrate memory onto the processor die



Currently

- DIMM modules place packaged die on both sides of a printed circuit board
- Error Correcting versions add an extra chip
- DRAM stacks won't be 8 deep, so can't add a chip

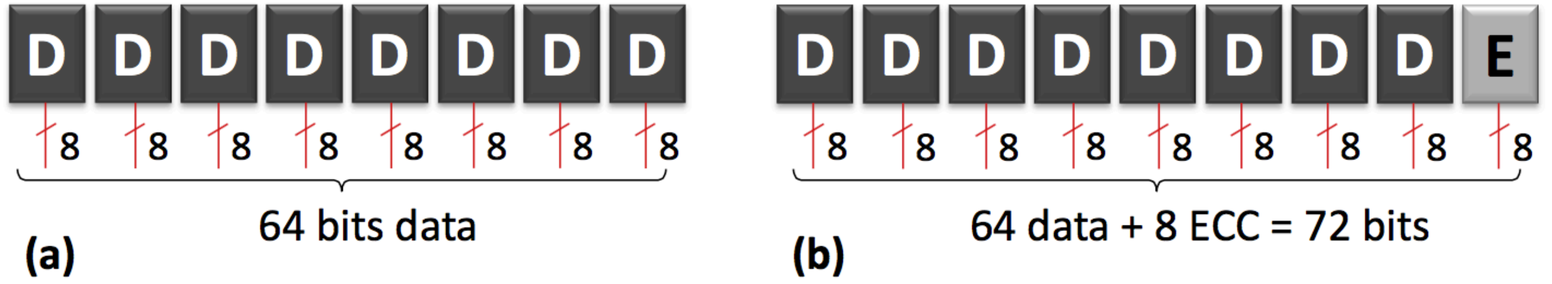


Figure 1: Organization of DRAM chips on a DIMM (one side, x8 chips) for (a) non-ECC DRAM, and (b) ECC DRAM.

Bad Idea

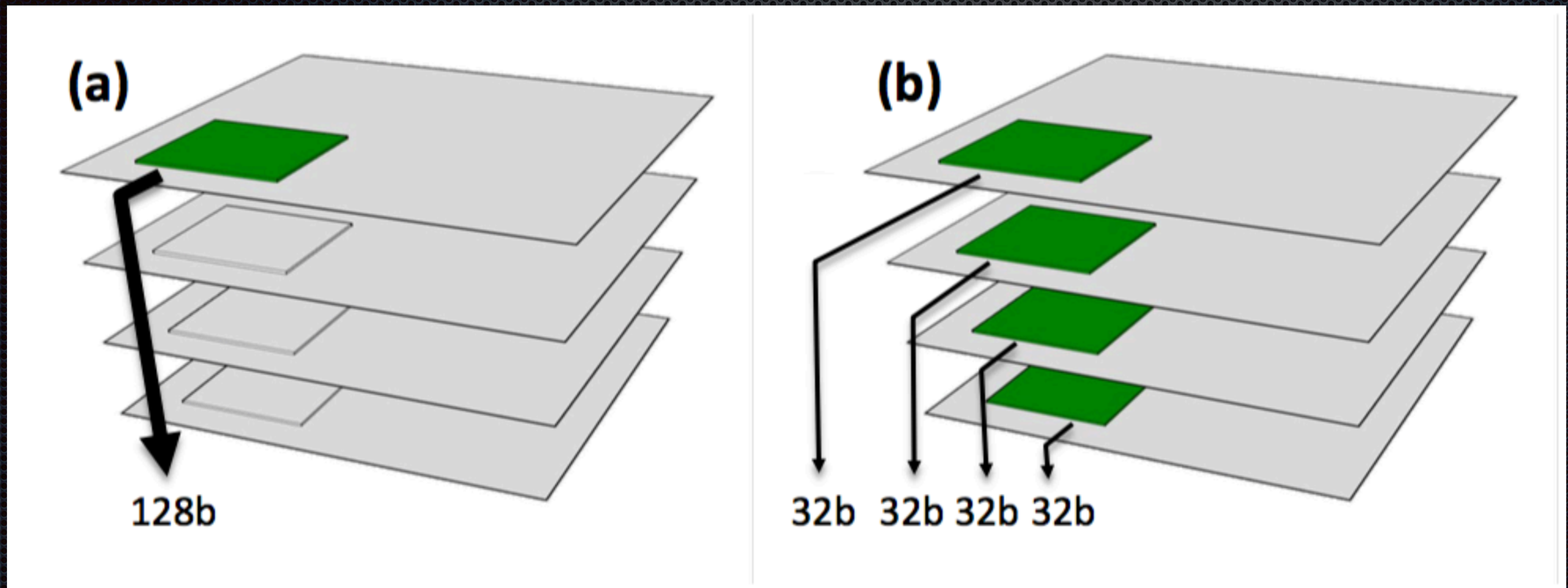


Figure 2: Reading data from a die-stacked DRAM with (a) all data delivered from a single bank from a single layer (similar to JEDEC Wide I/O [11]), and (b) data distributed across banks from four layers.

Why?

- Each chip already has multiple banks that can be read in parallel to achieve the desired bandwidth
- Different chips have slightly different timing
- More power is needed to activate all chips at once
- Constrains height of stack
- Prevents disabling defective chips to keep running

Error Modes

- The usual suspects: transient bit faults, row/column/bank faults, full chip failure
- Memory stack adds
  - Failed TSVs (expansion/contraction can break them)
  - Increased NBTI, HCI, and metal migration due to higher temperatures
- Stacks on top of processors make the whole assembly more costly to replace and decrease MTBF

DRAM as Cache

- The stacked DRAM on the processor is limited in size
  - But it is very fast
- Other memory will be needed externally (slower)
  - Can be very big
- If it walks like a cache...

Direct mapped is preferred

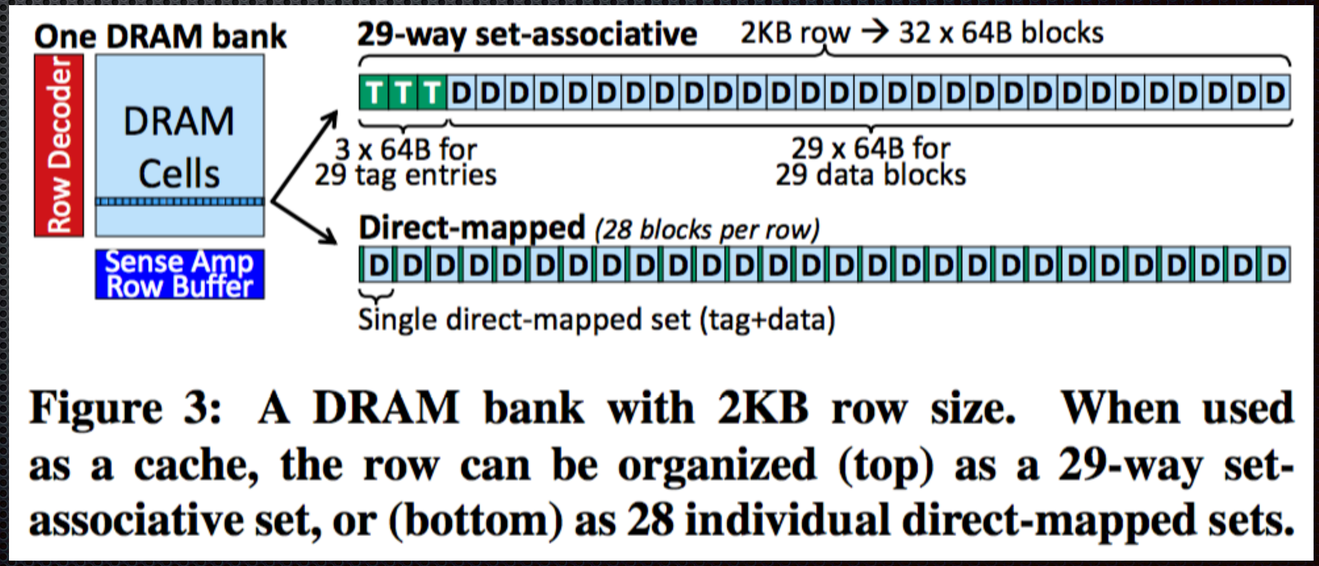
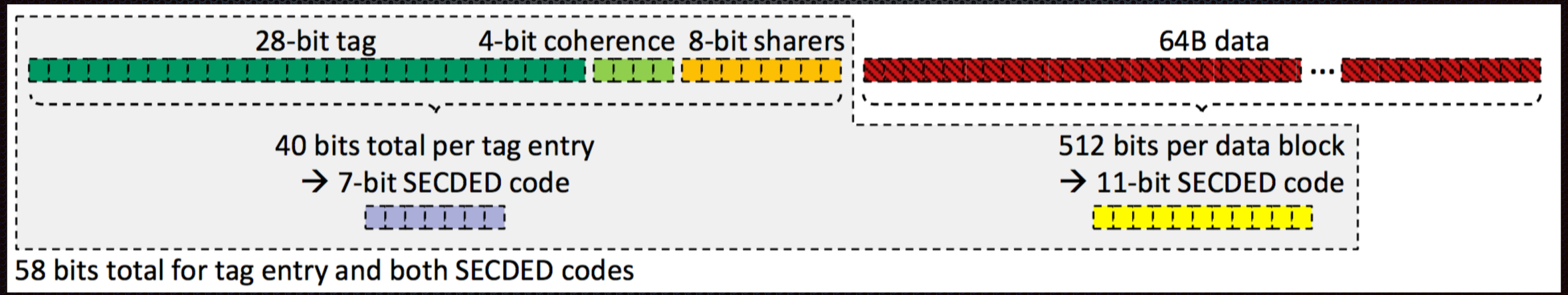


Figure 3: A DRAM bank with 2KB row size. When used as a cache, the row can be organized (top) as a 29-way set-associative set, or (bottom) as 28 individual direct-mapped sets.

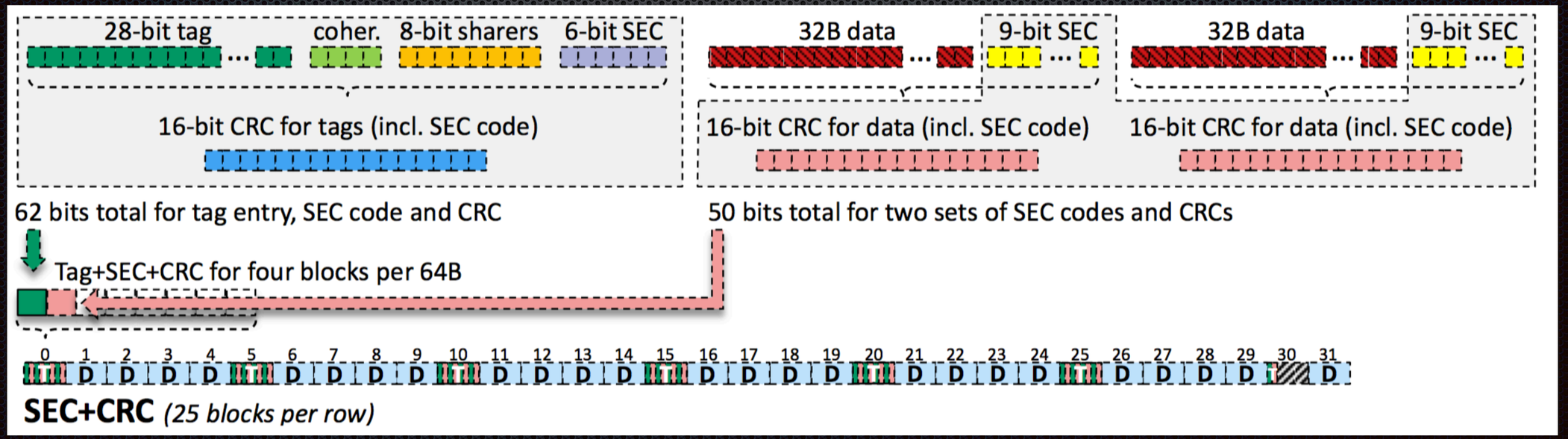
Error Correction

- Tags, coherence state, and directory need their own correction
- Data also needs correction
- For 64B of data, 5B of tag, that means 18 bits for correction (3.2% overhead)



Correction and Detection

- Adding cyclic redundancy check increases overhead
- Down from 28 to 25 blocks per row



Row Decoder Fault

- Won’t trigger an error (the wrong row is self consistent)
- Need to check the index of what gets fetched

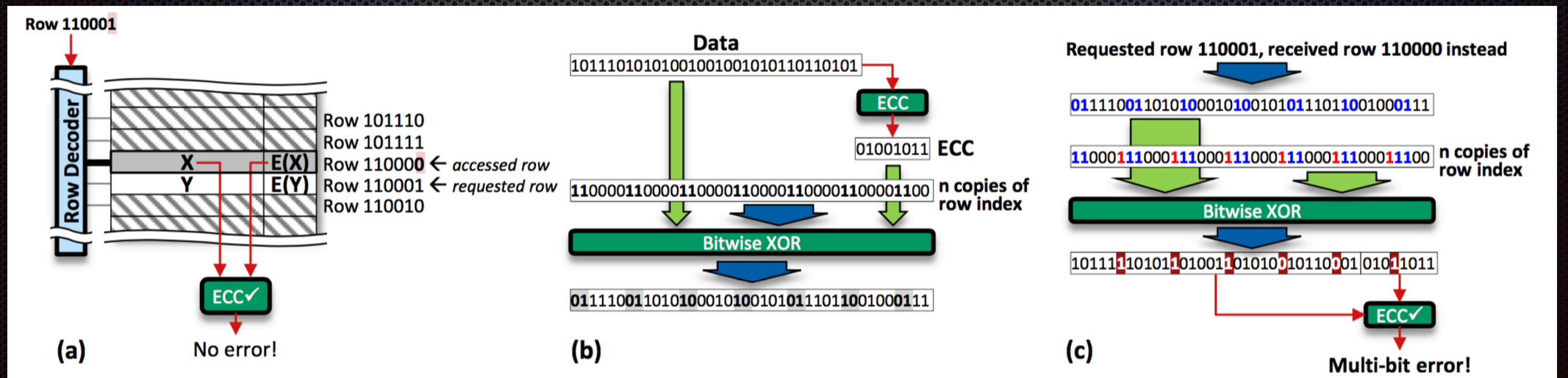


Figure 6: (a) Row decoder error that selects the incorrect row, which is undetectable using within-row ECC. (b) Process for folding in the row index (row 110000_2), and (c) usage of the folded row index to detect a row-decoder error.

Other Multibit Errors

- Clean data is already backed up
- Duplicate on write copies dirty data to enable recovery

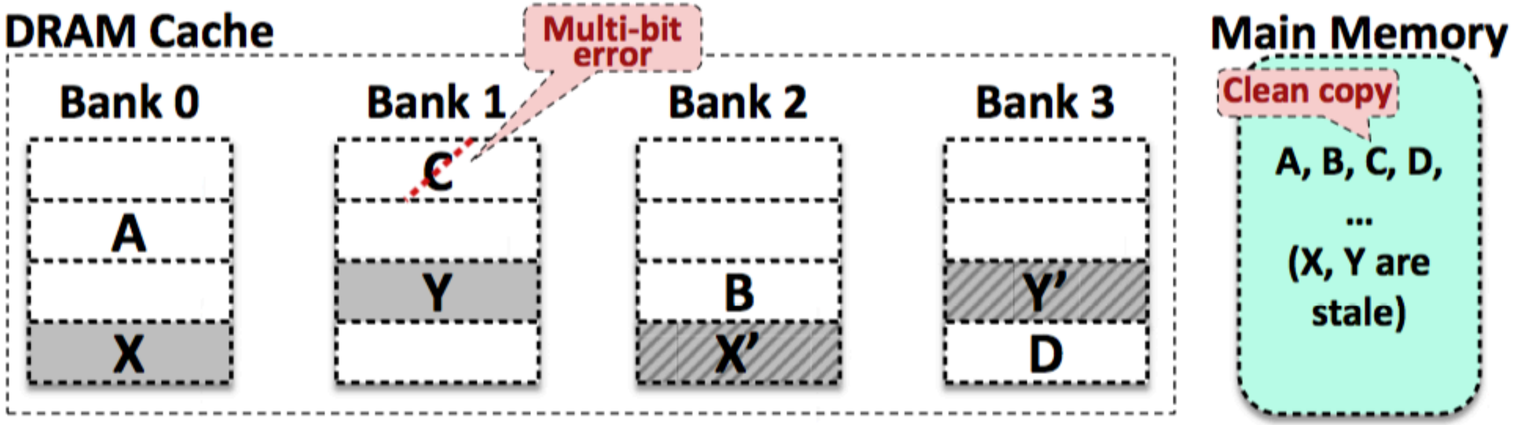


Figure 7: Example DRAM cache contents in which clean data are backed-up by main memory, but dirty data are duplicated into other banks.

Simulation

- Quad-core
- 4-way 32KB L1I and L1D (2 cycles)
- 16-way shared, unified L2 4MB (24 cycles)
- 128 MB DDR 3.2GHz, 128 bit channel
- 1.6GHz DRAM with 64 bit channel (unlimited size)

Workloads

Mix	Workloads	Group
WL-1	4 × mcf	4×H (rate)
WL-2	4 × leslie3d	4×H (rate)
WL-3	mcf-lbm-milc-libquantum	4×H
WL-4	mcf-lbm-libquantum-leslie3d	4×H
WL-5	libquantum-mcf-milc-leslie3d	4×H
WL-6	mcf-lbm-libquantum-soplex	3×H + 1×M
WL-7	mcf-milc-wrf-soplex	2×H + 2×M
WL-8	lbm-leslie3d-wrf-soplex	2×H + 2×M
WL-9	milc-leslie3d-GemsFDTD-astar	2×H + 2×M
WL-10	libquantum-bwaves-wrf-astar	1×H + 3×M
WL-11	bwaves-wrf-soplex-astar	1×H + 3×M
WL-12	bwaves-wrf-soplex-GemsFDTD	4×M

Table 3: Multi-programmed workloads.

H = High memory impact, M = medium

Performance

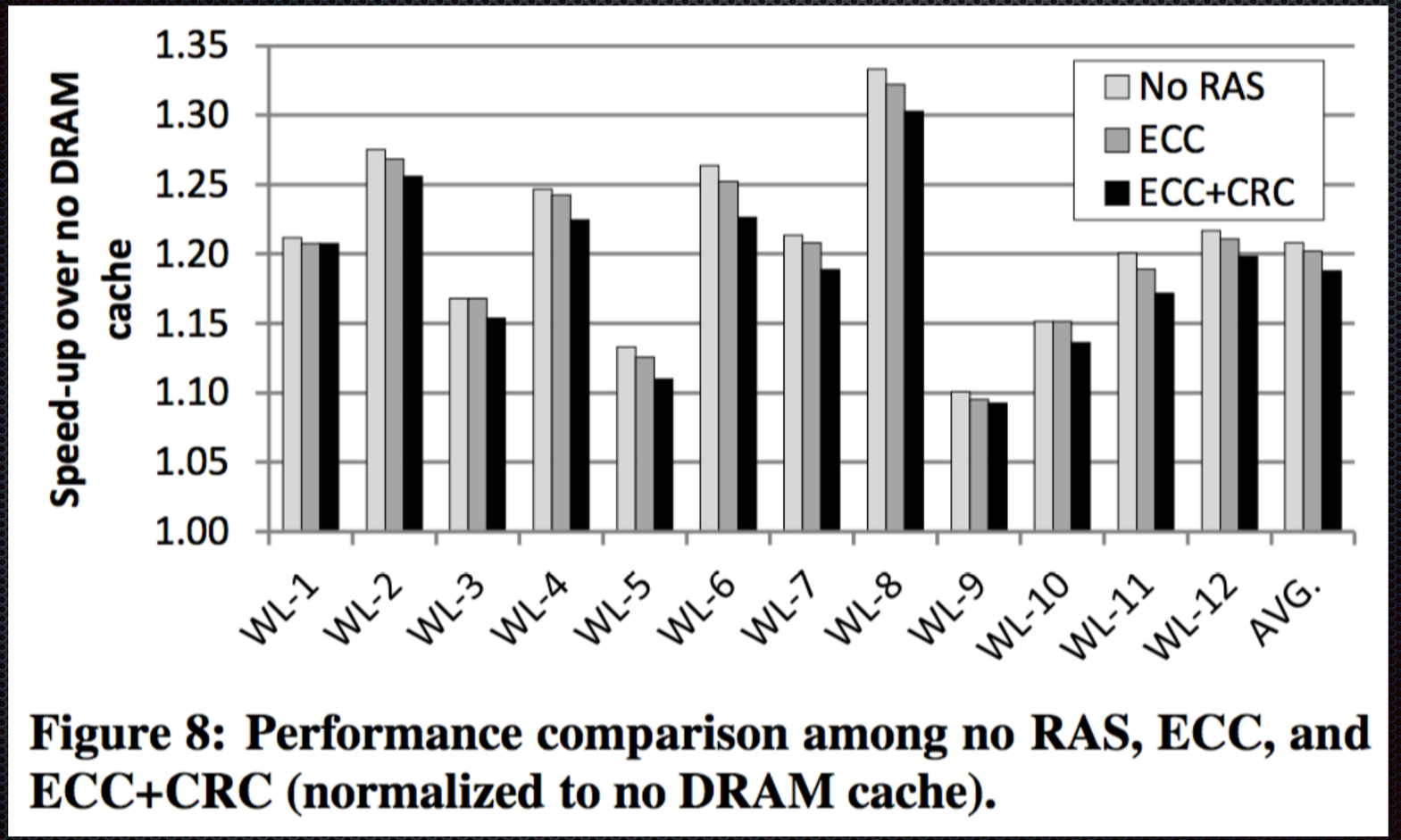


Figure 8: Performance comparison among no RAS, ECC, and ECC+CRC (normalized to no DRAM cache).

Performance
(heavier ECC, checking, and duplicating)

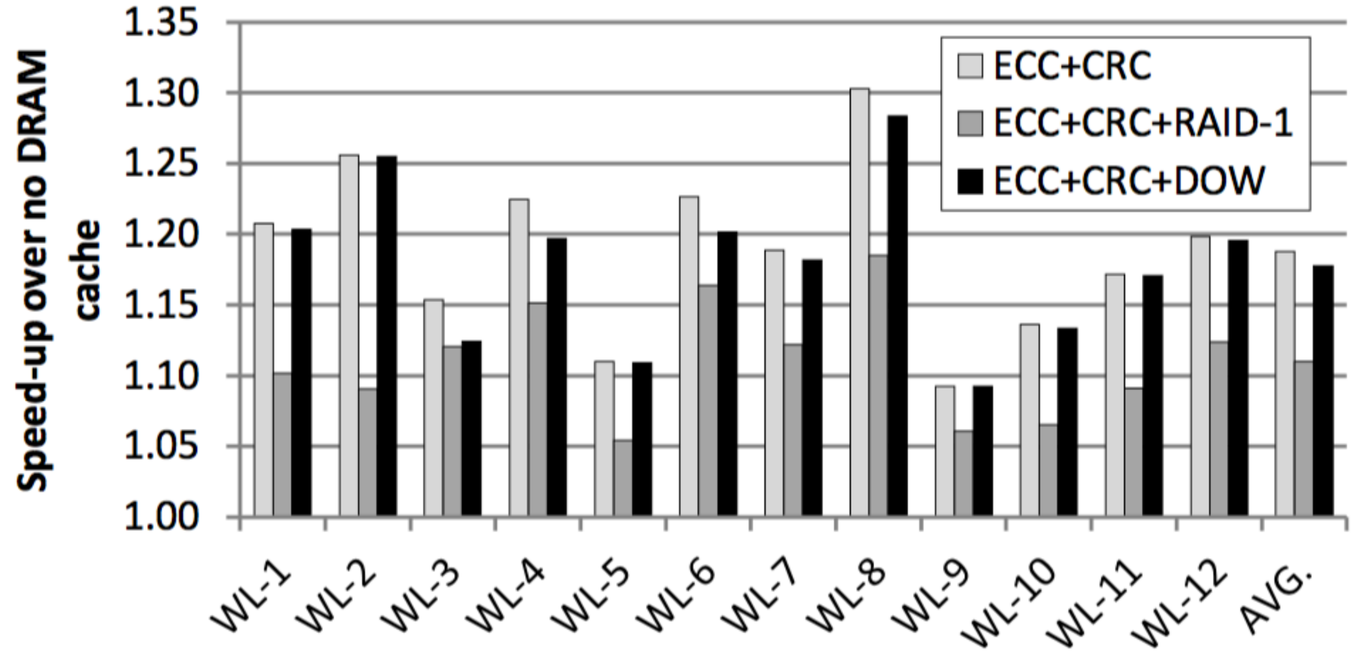


Figure 9: Performance comparison between fine-grain (ECC+CRC) and coarse-grain (ECC+CRC+RAID-1 and ECC+CRC+DOW) schemes (normalized to the performance without the DRAM cache).

Cost of DOW

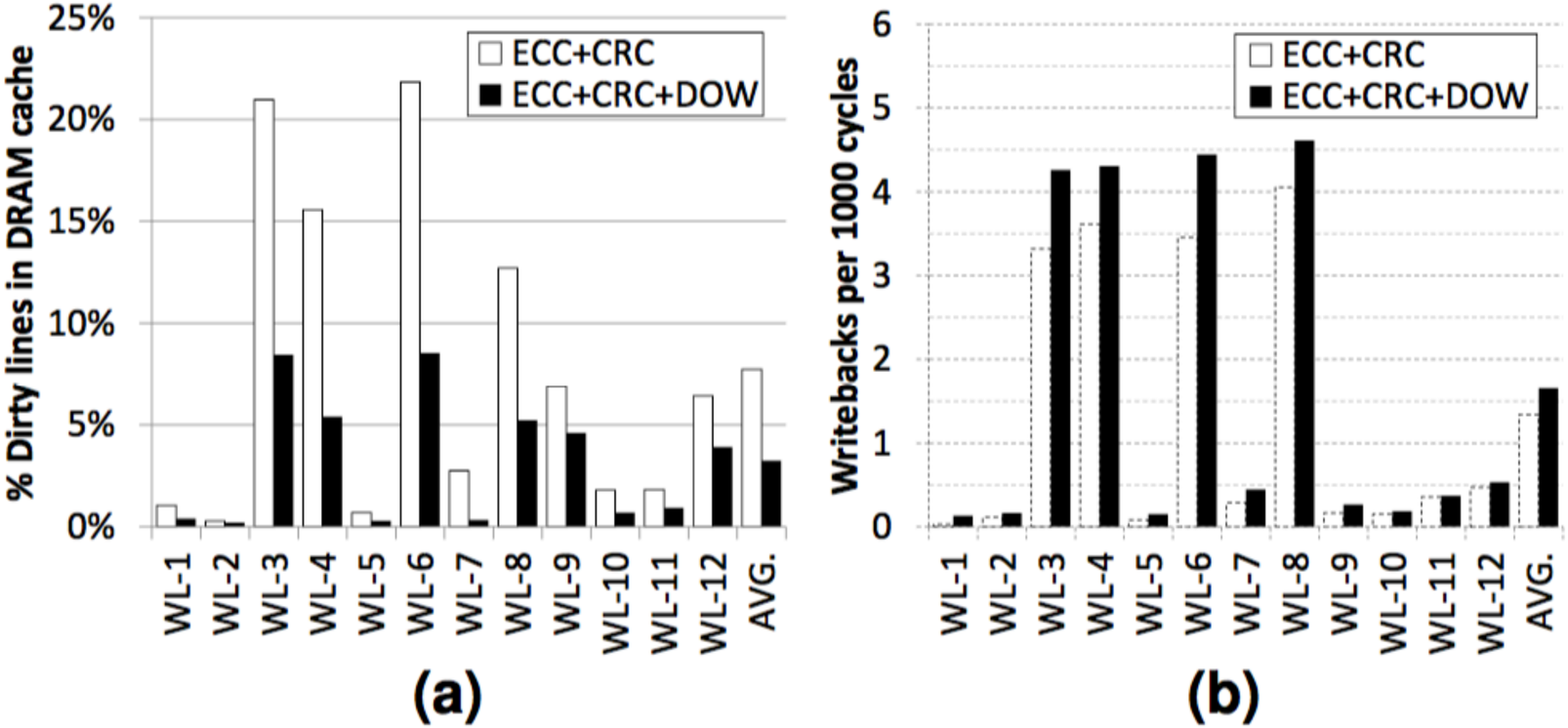


Figure 11: Impact of DOW on (a) the amount of dirty lines in the DRAM cache and (b) the writeback traffic from the DRAM cache.

Detect/Correct Rates

	Detection			
Failure Mode	**No RAS**	**ECC Only**	**ECC+CRC**	**DOW**
Single Bit	0%	100%	100%	100%
Column	0%	85%	99.9993%	99.9993%
Row	0%	50%	99.9993%	99.9993%
Bank	0%	50%	99.9993%	99.9993%

Table 5: Detection coverage for each technique.

	Correction			
Failure Mode	**No RAS**	**ECC Only**	**ECC+CRC**	**DOW**
Single Bit	0%	100%	100%	100%
Column	0%	85%	▷ 85% ◁	99.9993%
Row	0%	▷ 0% ◁	▷ 0% ◁	99.9993%
Bank	0%	▷ 0% ◁	▷ 0% ◁	99.9993%

Table 6: Correction coverage for each technique. Cases where correction coverage differs from detection coverage (Table 5) are marked with ▷◁.

Discussion

Density Tradeoffs of Non-Volatile Memory as a Replacement for SRAM based Last Level Cache

Korgaonkar, et.al., ISCA 2018

Back to the Wall

- 23 years later, still concerned about the memory wall
- Now due more to bandwidth than latency (multicores need a lot of memory)
- On-chip cache has much higher bandwidth than off-chip RAM
- SRAM density isn’t high enough for bigger Last Level Cache (LLC)
  - Static leakage power increases with smaller feature sizes
  - DRAM needs refresh, also doesn’t scale down as much as needed

Spin-Torque Transfer RAM

- Non-volatile, non-destructive read-out
- Additional masks/layers on top of standard CMOS (easier than DRAM)
- Scales below 10nm
- Single transistor with ferro-magnetic junction layers (fixed, barrier, free) above drain
- Low leakage power, no refresh, write is slow and requires high current

LLC (not the legal kind)

- Typically level 3 or 4 unified cache
- Example: Haswell Xeon 18-core has
  - 32KB L1D, 32KB L1I, 256 KB L2 per core (1.15 MB L1, 4.5MB L2)
  - Shared 45MB L3
- Core i9 with 8 cores has 16MB L3
- Big enough for working set of a single core, or a few lightly loaded cores
- With 18 cores, will still see high miss rate for diverse job mix

LLC meets STT

- STT has high write latency
  - For write-heavy workloads, can create a bottleneck
- Buffering queues can handle bursts, but not sustained dense writes
- Some proposals for changing device characteristics, write power, quality of writes, but these reduce density or reliability
- Prior work on fusing redundant or rapid-replacement writes to reduce number reaching LLC still leaves room for improvement

Two Proposed Techniques

- Congestion-aware bypass
  - Send writes directly to RAM when congested (reduces hit rate)
- Virtual hybrid cache
  - Change

Pros and Cons

- Increasing LLC from 4MB to 16MB gives a 23% performance increase
  - Note that many chips now have 16MB or more
  - Not clear that, e.g., increasing from 45MB to 90MB has equal effect
- If STT is only twice as dense, 8MB gives 15%
- But if latency goes up by 20ns, then ends up with 15% loss

Inclusive/Exclusive LLC

- Inclusive gets writes from LLC miss loads, and L2 dirty evicts
- Exclusive gets L2 clean and dirty evicts, plus deallocate on L2 hit
  - An advantage of exclusive is greater overall effective cache size
  - A disadvantage is increased coherence traffic to the L2 cache
  - When shared, LLC can only deallocate if all L2s get copies or if L2s use a more complex protocol, such a MOESI, that updates between caches

Write Bypassing

- Use a subset of L2 lines to sample liveness (an instruction at a location repeatedly requests a value from the line)
- Other lines also get a liveness score (not clear how)
- Liveness score accompanies eviction to LLC write queue
- Queue keeps a congestion score, used to select liveness threshold
  - Queued write with a liveness score below threshold are bypassed

Virtual Hybrid Cache

- Part of L2 (2 ways) and LLC are used to coalesce writes to lines that have frequent updates (dirty fills)
- Change replacement policy to keep frequent updates in 2 L2 ways for longer (reduces dirty writes)
- Change exclusive LLC policy to not immediately deallocate reads to L2 so that they are inclusive for some period (reduces clean writes)

Simulation

- Quad core x86 with 32KB L1D, L1I, 256KB L2, shared 4MB cache
- Assume STT of 8MB with 20ns additional latency (128K lines)
- Construct SPEC workload misses with high write frequency
- 250M instructions per core, 1B instructions total

Methodology Question

- Assuming typical 2 reads and 1 write per 10 instructions, 300,000,000 accesses
- Assuming 10% L1D miss rate and 2% L2 miss rate, about 600K accesses reach LLC
- Statistical significance requires about 100 accesses per line
- With 128K lines, 600K accesses can at most touch each line 4.7 times

Results

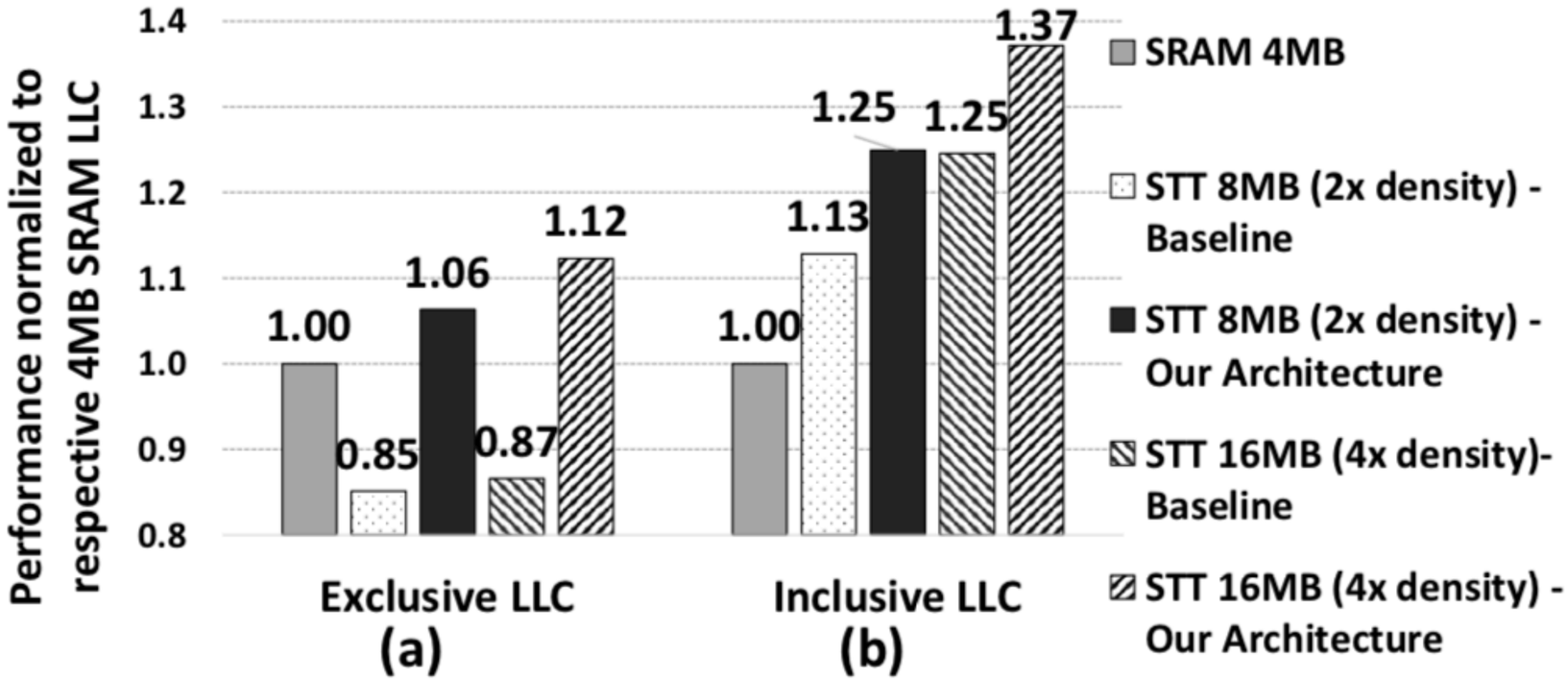


Figure 9: Performance compared to 4MB SRAM LLC

Bypassing Impact

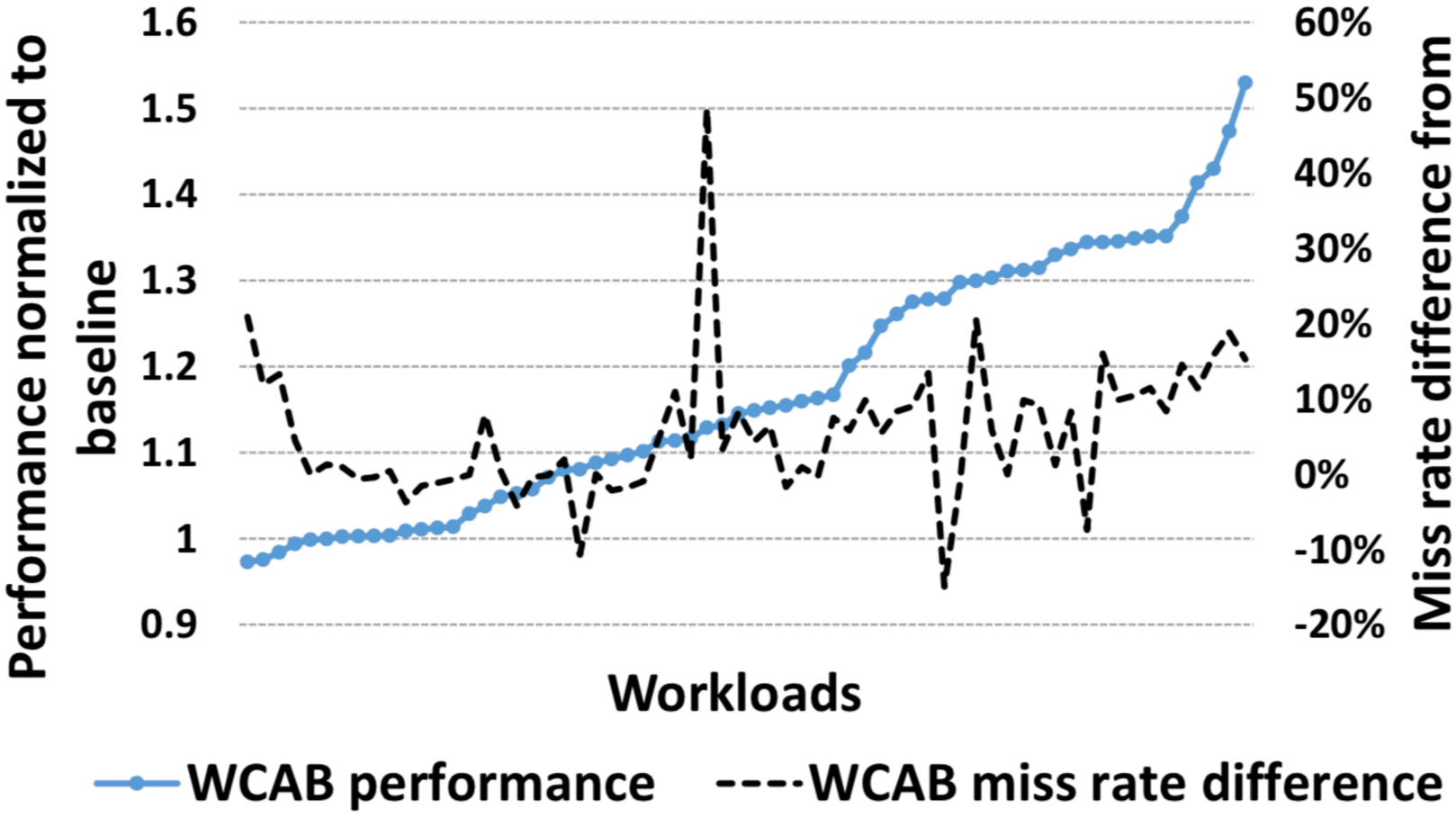


Figure 11: Performance vs. miss rate difference of WCAB.

Movement Policy Impact

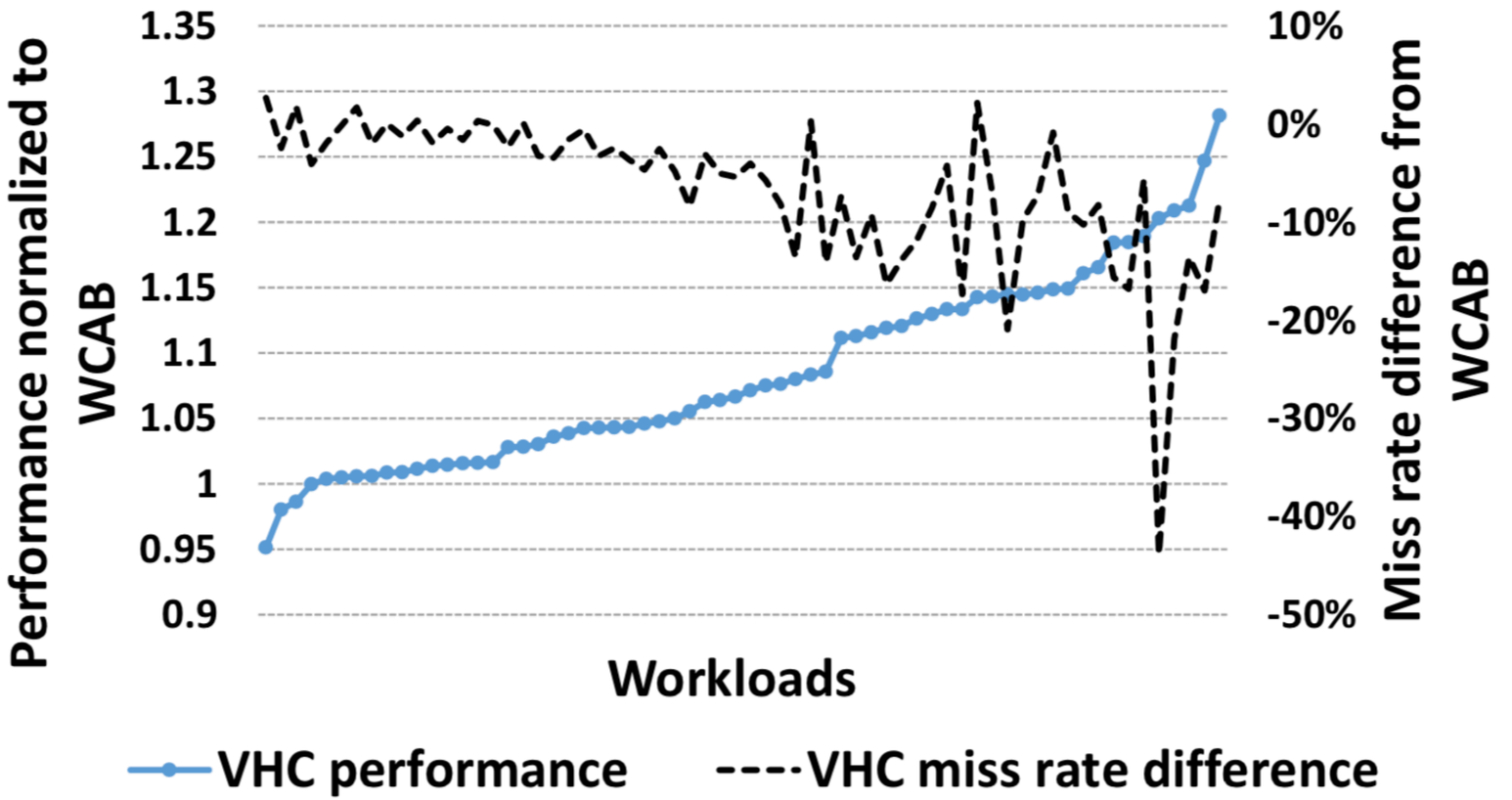


Figure 12: Performance vs. miss rate difference of VHC on top of WCAB.

On More Realistic Workloads

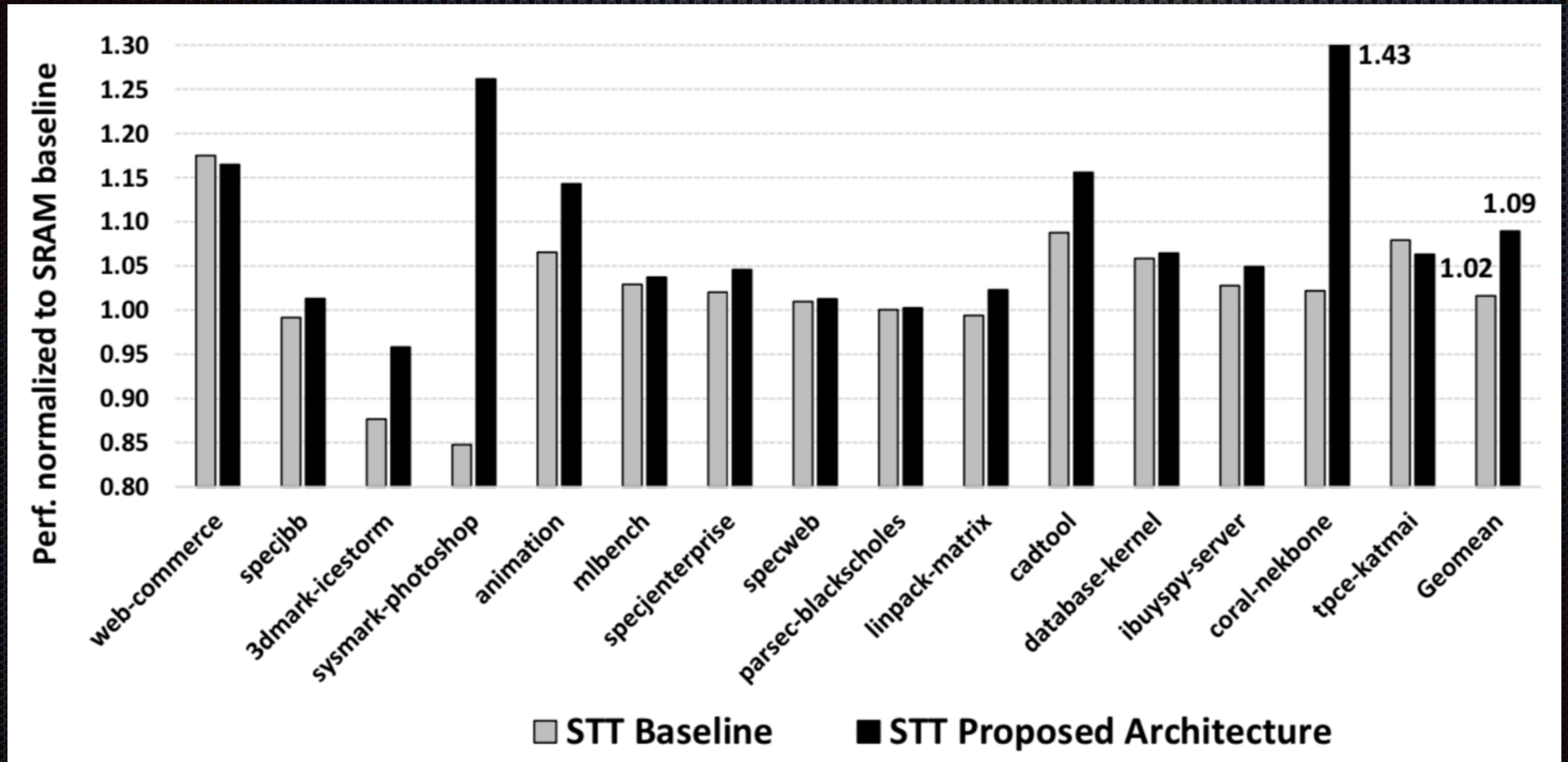


Figure 20: Evaluation on industry workloads

Discussion