

Cache Design 2

CS/COE 1541 (Fall 2020)
Wonsun Ahn

Cache Design Parameter 5: Write-Through vs. Write-Back

Writes and Cache Consistency

- Assume **&x** is 111010_2 , and $x == 24$ initially

lw t0, &x

000

addi t0, t0, 1 # x++

001

sw t0, &x

010

- How will the **lw** change the cache?

011

- How will the **sw** change the cache?

100

- Uh oh, now the cache is **inconsistent**.
(Memory still has the old value 24.)

101

110

111

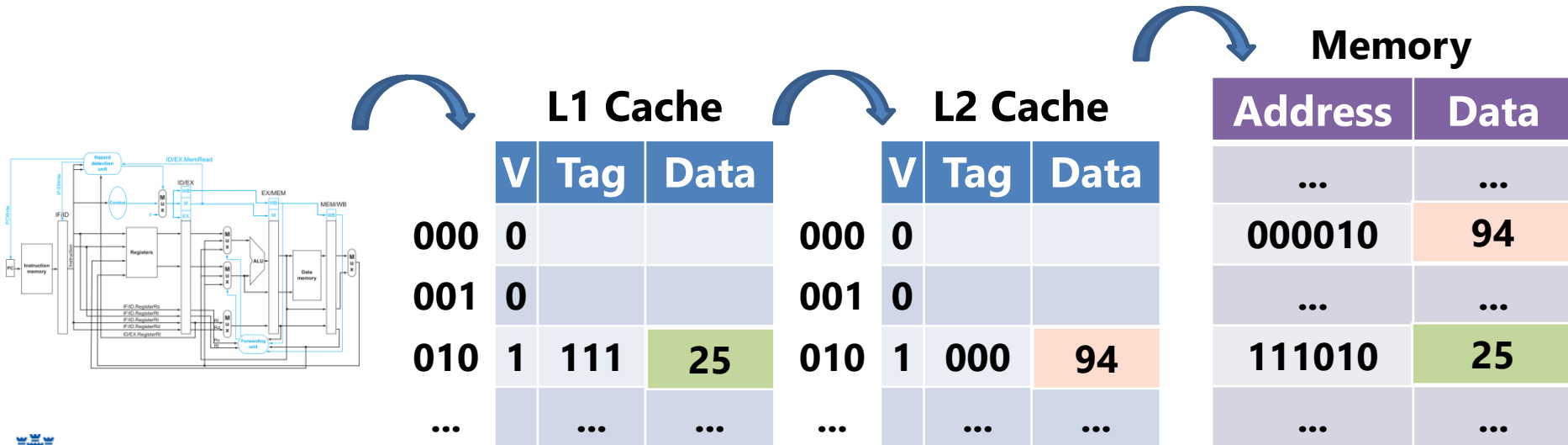
V	Tag	Data
0		
0		
1	111	25
0		
0		
0		
0		
0		

- How can we solve this? Two policies:

- Write-through**: Propagate write all the way through memory
- Write-back**: Write back cache block when it is evicted from cache

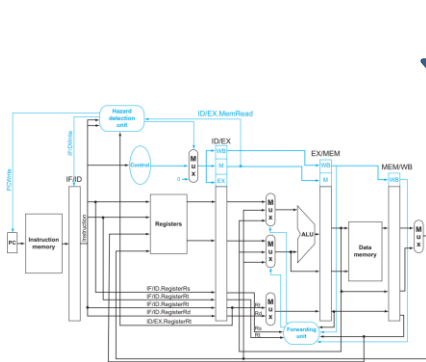
Policy 1: Write-through

- Write-through:
 - If write hit, update cache block in current cache
 - Propagate write to lower memory to update it as well
- What happens if we write **25** to address **111010₂**?
- What happens if we write **94** to address **000010₂**?
- Caches are kept consistent at all points of time!



Write-through: Reads

- What happens if we read from address **000010₂**?
 - We can just discard the conflicting cache block **111010₂**
 - It's just an extra copy of the same data
- Note how we allocate lines only on reads
 - We do not allocate lines on writes
 - This policy is called **no write allocate**



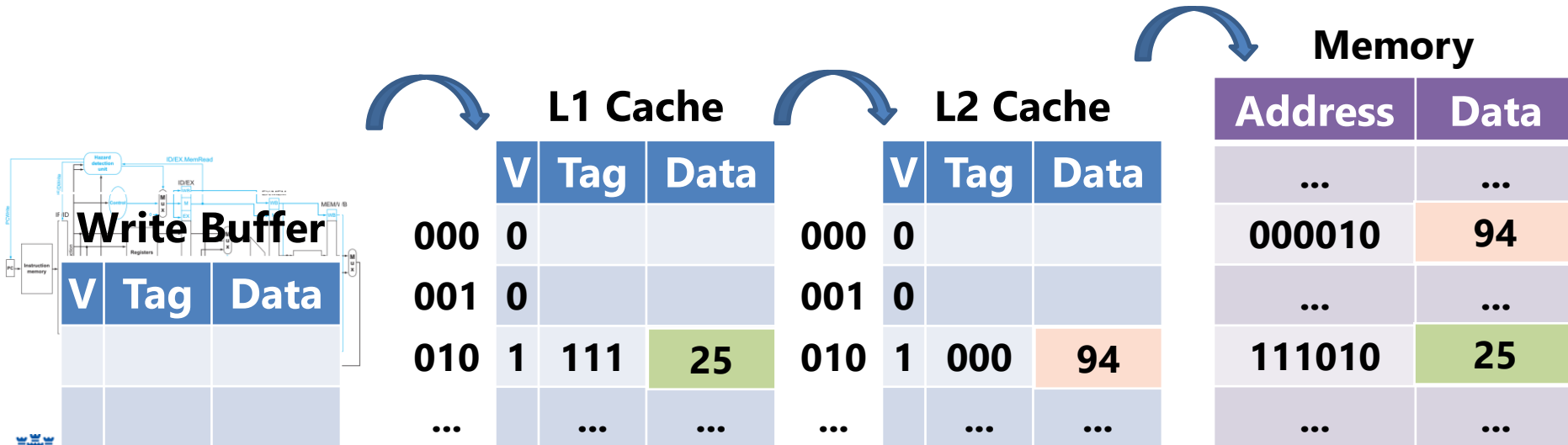
	L1 Cache				L2 Cache		
	V	Tag	Data		V	Tag	Data
000	0			000	0		
001	0			001	0		
010	1	000	94	010	1	000	94
...	

Memory

Address	Data
...	...
000010	94
...	...
111010	25
...	...

Write-through: Drawbacks

- Drawback: **Long write delays** regardless of hit or miss
 - Must propagate write all the way to DRAM memory regardless
- Solution: **Write buffer** maintaining pending writes
 - CPU gets on with work after moving pending write to write buffer
 - But does the write buffer solve all problems?

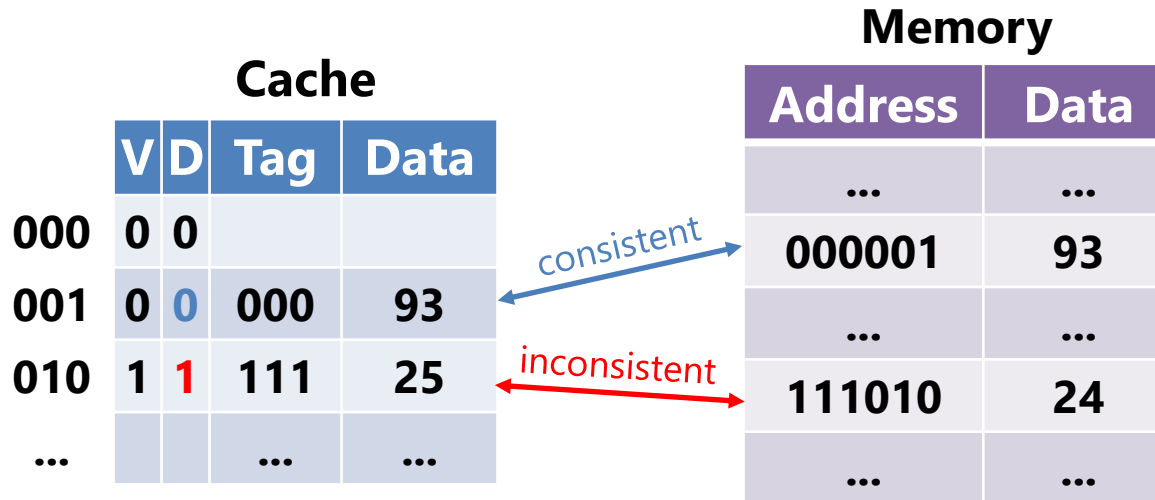


Write-through: Drawbacks

- The write buffer does not solve all problems.
 1. Write buffer must be **very big** to store all pending writes
 - May take more than 100 cycles for write to propagate to memory
 - Write buffer is always checked before L1\$ → adds to **hit time**
 2. Write buffer does not solve **bandwidth** problems
 - If memory bandwidth < rate of writes in program, write buffer will fill up quickly, no matter how big it is
- Impractical to write-through all the way to memory
 - Typically only L1 caches are write-through, if any
- We need another strategy that is not so bandwidth-intensive

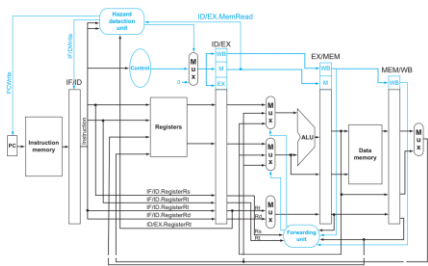
Policy 2: Write-back

- Write-back:
 - **Dirty** block: a block that is temporarily inconsistent with memory
 - Update cache block in current cache only, marking it dirty
 - Write back dirty block to memory when it is evicted from cache
- A **dirty bit** is added to the cache block metadata (marked "D")
 - Block **000001**₂ is clean → can be discarded on eviction
 - Block **111010**₂ is dirty → needs to be written back on eviction



Policy 2: Write-back

- What happens if we write **25** to address **111010₂**?



Write Buffer

V	Tag	Data

000
001
010
...

V	D	Tag	Data
0	0		
0	0		
1	0	111	24
	

L1 Cache

000
001
010
...

V	D	Tag	Data
0	0		
0	0		
1	0	000	93
	

L2 Cache

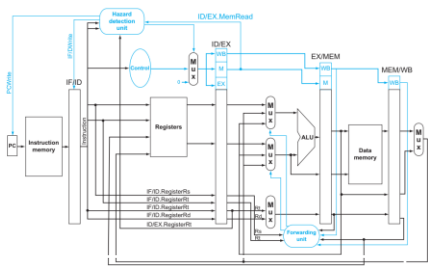


Memory

Address	Data
...	...
000010	93
...	...
111010	24
...	...

Policy 2: Write-back

- What happens if we write **25** to address **111010₂**?
 - Hit! Update cache block and mark it dirty.



Write Buffer

V	Tag	Data
000	0 0	
001	0 0	
010	1 1	25
...

V	D	Tag	Data
000	0 0		
001	0 0		
010	1 1	111	25
...

L1 Cache

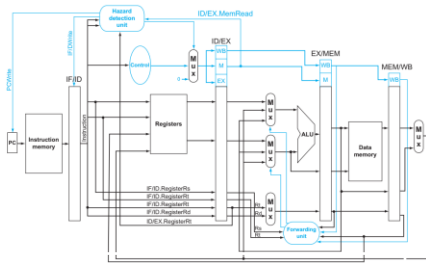
V	D	Tag	Data
000	0 0		
001	0 0		
010	1 0	000	93
...

L2 Cache

Memory	
Address	Data
...	...
000010	93
...	...
111010	24
...	...

Policy 2: Write-back

- What happens if we write **25** to address **111010₂**?
 - Hit! Update cache block and mark it dirty.
- What happens if we write **94** to address **000010₂**?
 - L1 Miss! Before bringing in **94**, must evict and write-back **25**
(Note: can discard L2 cache block with **93** since it's not dirty)



Write Buffer

V	Tag	Data

000
001
010
...

V	D	Tag	Data
0	0		
0	0		
1	1	111	25
	

L1 Cache

000
001
010
...

V	D	Tag	Data
0	0		
0	0		
1	0	000	93
	

L2 Cache

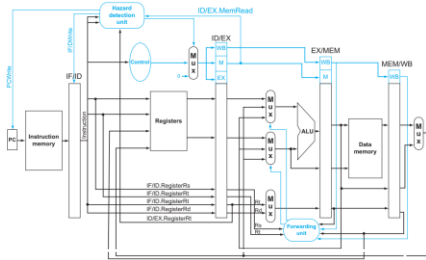


Memory

Address	Data
...	...
000010	93
...	...
111010	24
...	...

Policy 2: Write-back

- What happens if we write **25** to address **111010₂**?
 - Hit! Update cache block and mark it dirty.
- What happens if we write **94** to address **000010₂**?
 - L1 Miss! Before bringing in **94**, must evict and write-back **25**
(Note: can discard L2 cache block with **93** since it's not dirty)
 - L2 Miss! Must evict and write-back **25** again to make space for **94**



Write Buffer

V	Tag	Data

	V	D	Tag	Data
000	0	0		
001	0	0		
010	0	0		
...		

L1 Cache

	V	D	Tag	Data
000	0	0		
001	0	0		
010	1	1	111	25
...		

L2 Cache

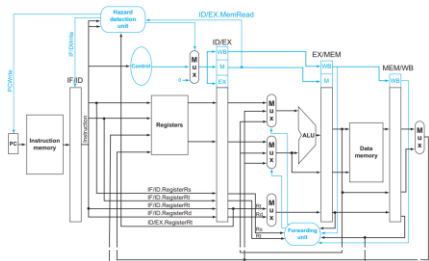


Memory

Address	Data
...	...
000010	93
...	...
111010	25
...	...

Policy 2: Write-back

- What happens if we write **94** to address **000010₂**? (cont'd)
 - Now we have allocated the spaces for **94**, bring the block in!
 - And then, update the block and mark it dirty.
- Note how we always allocate blocks for writes on a write miss
 - This policy is called **write allocate**
 - Implication: still **reads memory** on write **misses** (but not hits)
 - Q: Why read memory when we'll overwrite it?



Write Buffer

V	Tag	Data
000	0	0
001	0	0
010	1	94
...

L1 Cache

V	D	Tag	Data
000	0	0	
001	0	0	
010	1	000	94
...

L2 Cache

V	D	Tag	Data
000	0	0	
001	0	0	
010	1	000	93
...

Memory

Address	Data
...	...
000010	93
...	...
111010	25
...	...

Write-back: Write allocate

- On a write miss, a cache block is always allocated in current cache
- That cache block must always be read in from lower memory
 - Before subsequently updating it with the write
 - Why the wasted effort?
- Because a block is multiple bytes, and you are updating just a few
 - Suppose a cache block is 8 bytes and you are writing to only 4

V	D	Tag	Data	
1	1		Upper 4 bytes (Not updated)	Lower 4 bytes (Updated)

- After allocate, the entire cache block is marked **valid**
 - **Upper 4 bytes** must be filled with valid data (from memory).
 - Unavoidable, unless you have a valid bit for each byte
(That means spending 1 bit for every 8 bits of data. Not done.)

Write-through vs. Write-back

- Advantages of write-through caches
 - Simpler to implement
 - Don't have to deal with block allocation on write misses
 - Don't have to deal with write-backs on block eviction
- Advantages of write-back caches
 - Lower bandwidth requirements
 - Lower memory accessed only on cache misses
(Unlike write-through which propagates all writes to memory)
 - Cache misses are typically a small percentage of all accesses
- L1 caches are sometimes write-through for simplicity
 - Plenty of bandwidth within chip to support this
- Last-level caches are almost always write-back
 - Long latency and low bandwidth to DRAM prohibits write-through

Cache Design Parameter 6: Unified vs. Split

Problem with Split Caches

- If cache is split into two (i-cache and d-cache)
 - Space cannot be flexibly allocated between data and code



If our working set looks like this – say, in a small loop that's accessing a large array – then we run out of data space.



If our working set looks like this – say, in a large function that's only using stack variables – then we run out of code space.

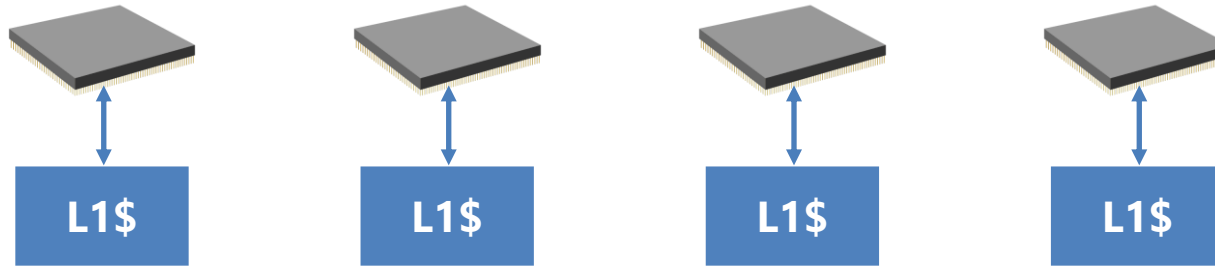
Impact of Unifying Cache

- The answer to the problem is to simply **unify** the cache into one
- $AMAT = \text{hit time} + (\text{miss rate} \times \text{miss penalty})$
- Impact of unifying cache on miss rate:
 - **Smaller miss rate** due to more flexible use of space
- Impact of unifying cache on hit time:
 - Potentially **longer hit time** due to structural hazard
 - With split caches, i-cache and d-cache can be accessed simultaneously
 - With unified cache, access request must wait until port is available
- **L1** cache is almost always **split**
 - Frequent accesses directly from pipeline trigger structural hazard often
- **Lower level** caches are almost always **unified**
 - Accesses are infrequent (filtered by L1), so structural hazards are rare

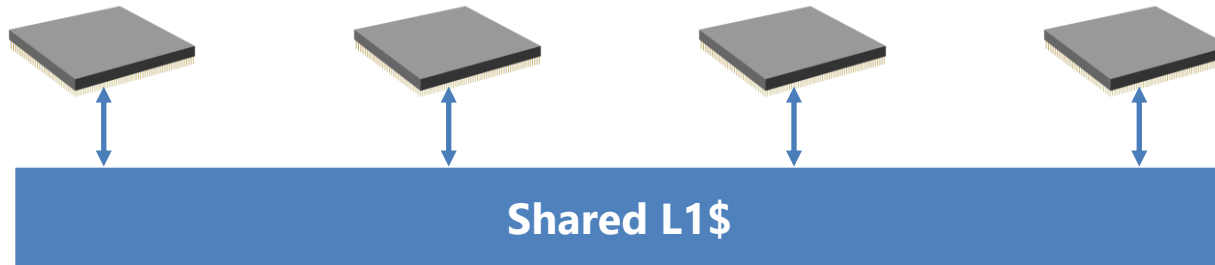
Cache Design Parameter 7: Private vs. Shared

Private vs. Shared Cache

- On a multi-core system, there are two ways to organize the cache
- **Private** caches: each core (processor) uses its own cache

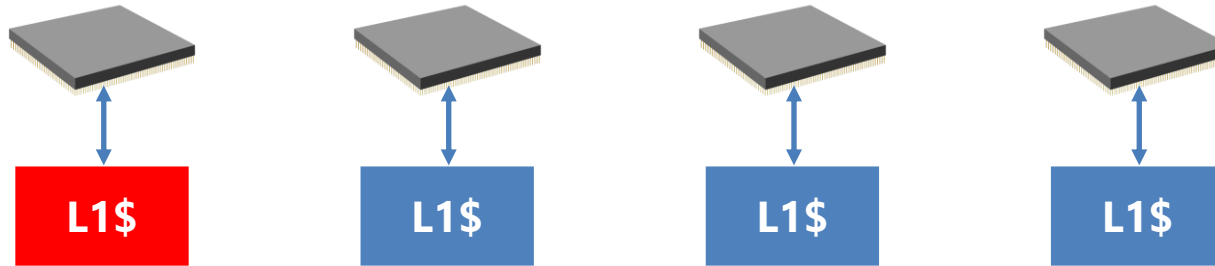


- **Shared** cache: all the cores share one big cache



Shared Cache can Use Space More Flexibly

- Suppose only 1st core is active and other cores are idle
 - How much cache space is available to 1st core? (Shown in **red**)
- **Private** caches: 1st core can only use its own private cache



- **Shared** cache: 1st core can use entire shared cache!

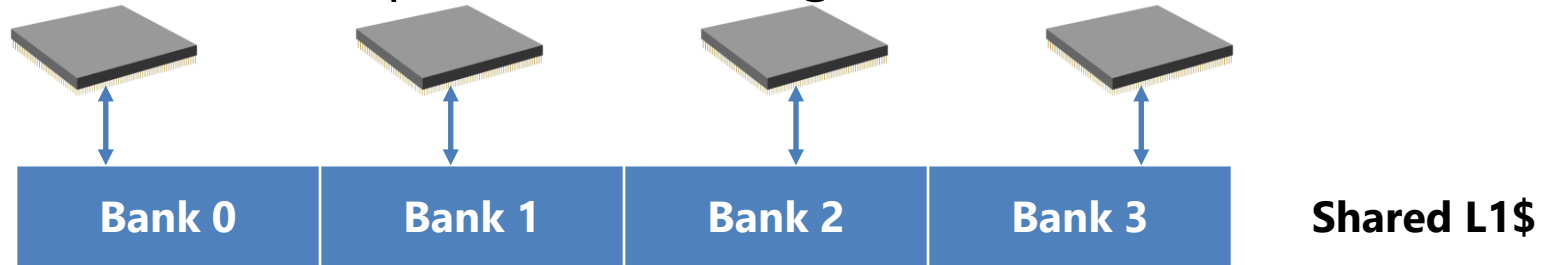


Banking: Solution to Structural Hazards

- Now what if all the cores are active at the same time?
 - Won't that cause structural hazards due to simultaneous access?



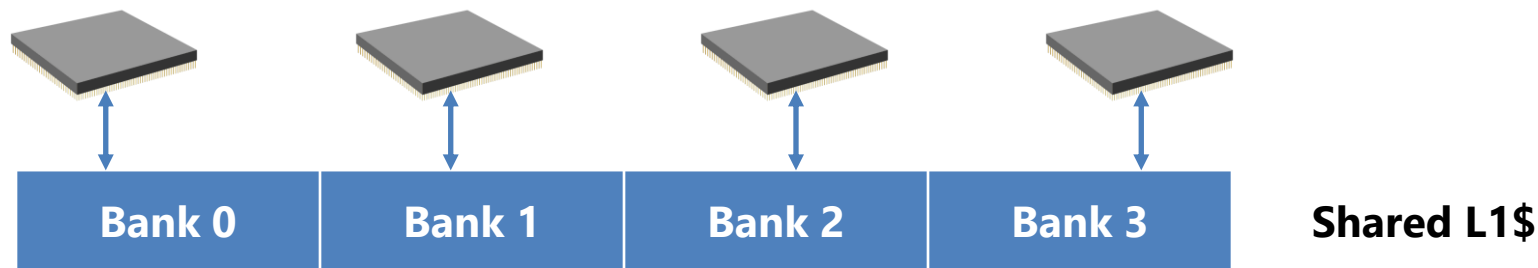
- Could add more ports, but adding **banks** is more cost effective



- Each bank has its own read / write port
- As long as two cores do not access same bank, no hazard!

Banking: Solution to Structural Hazards

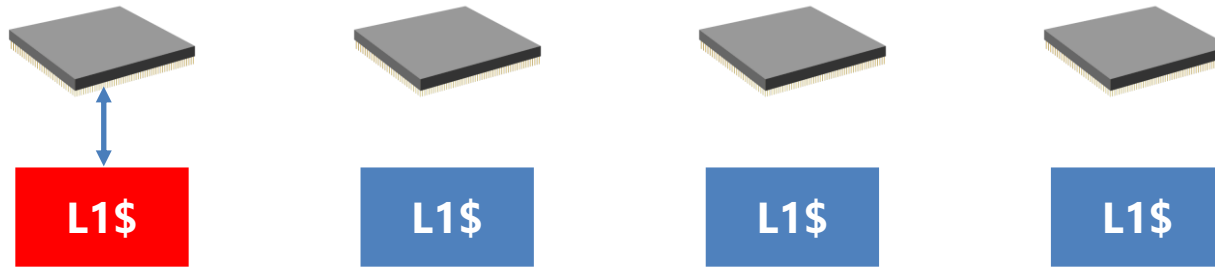
- Cache blocks are **interleaved** between banks



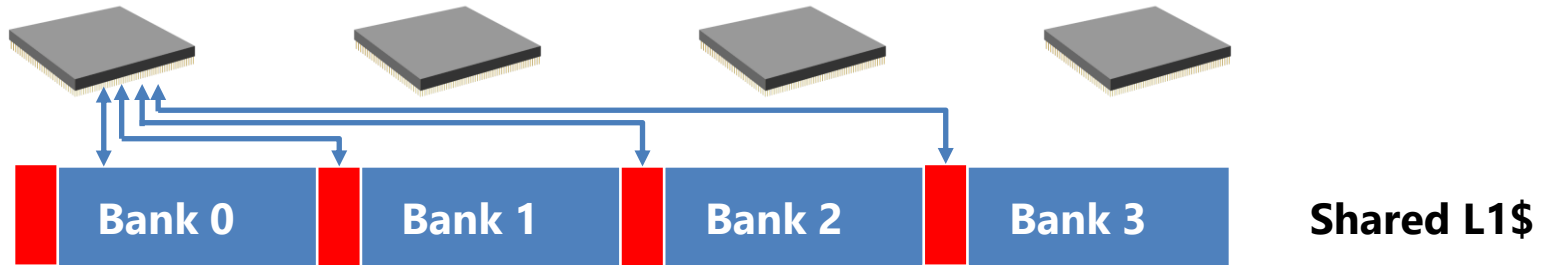
- Blocks 0, 4, 8 ... → Bank 0
- Blocks 1, 5, 9 ... → Bank 1
- Blocks 2, 6, 10 ... → Bank 2
- Blocks 3, 7, 11 ... → Bank 3
- That way, blocks are evenly distributed across banks
 - Causes cache accesses to also be distributed → less hazards

Shared Cache have Longer Access Times

- Again, suppose only 1st core is active and other cores are idle
 - The working set data is shown in **red**
- **Private** caches: entire working set data in nearby private cache

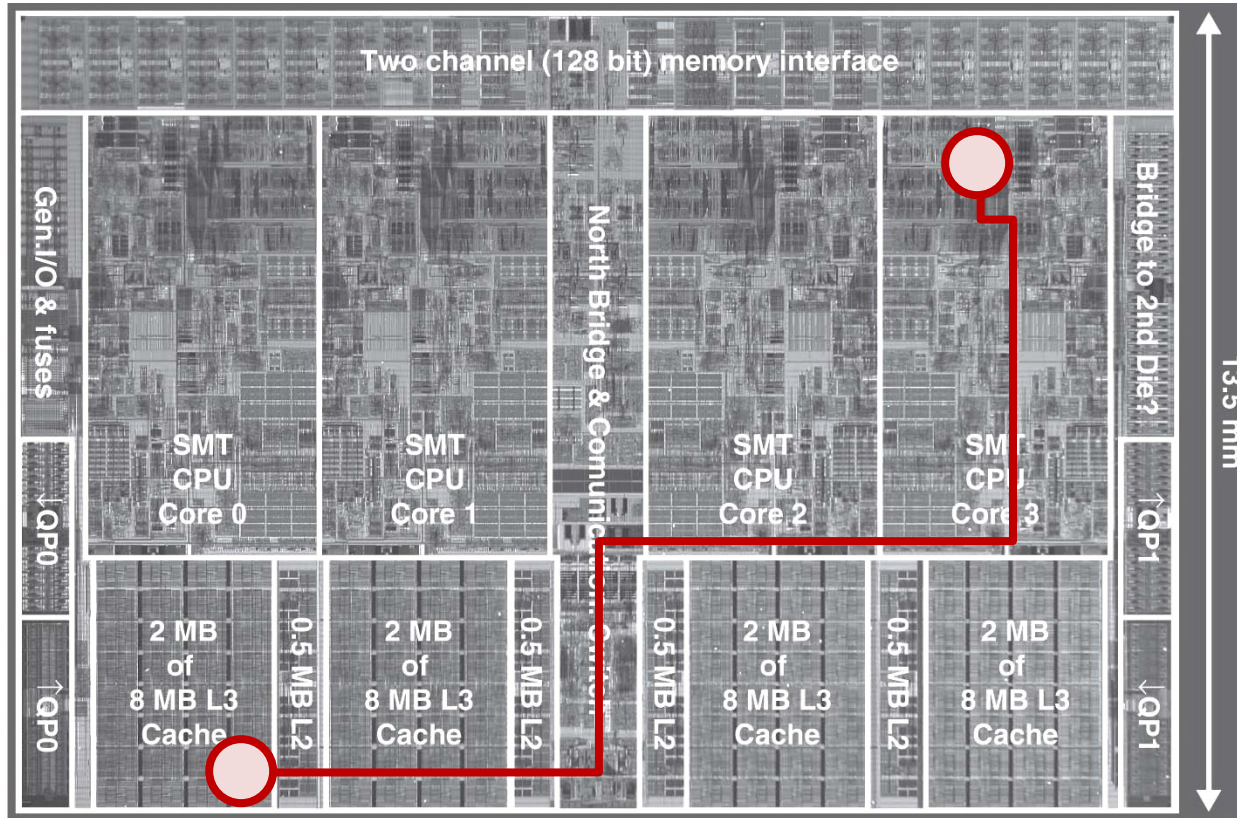


- **Shared** cache: data sometimes distributed to remote banks



Shared Cache have Longer Access Times

- Remember this picture?

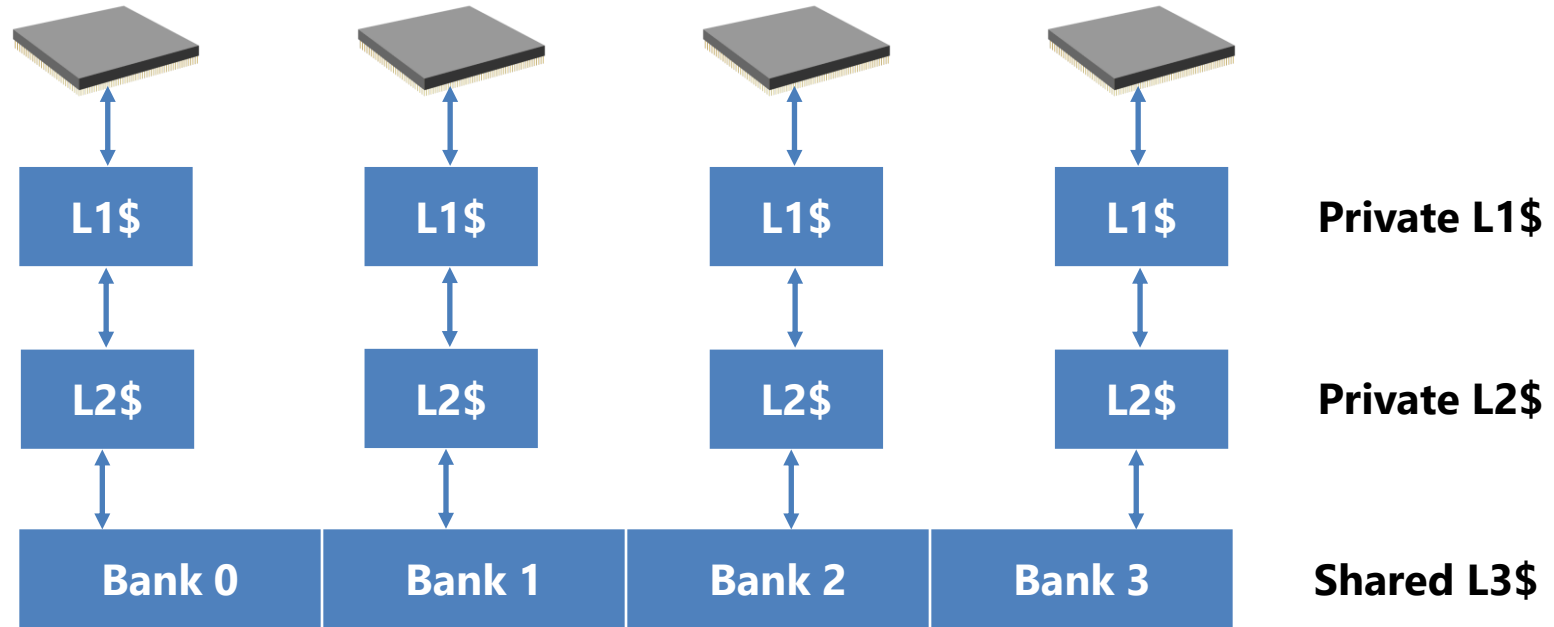


Impact of Shared Cache

- $AMAT = \text{hit time} + (\text{miss rate} \times \text{miss penalty})$
- Impact of shared cache on miss rate:
 - **Smaller miss rate** due to more flexible use of space
- Impact of shared cache on hit time:
 - **Longer hit time** due to sometimes having to access remote banks
- **L1** caches are almost always **private**
 - Hit time is important for L1. Cannot afford access to remote banks.
- **L3 (last level)** caches are almost always **shared**
 - Reducing miss rate is top priority to avoid DRAM access.

Cache Organization of Broadwell CPU

- This is the cache organization of Broadwell used in our Linux server



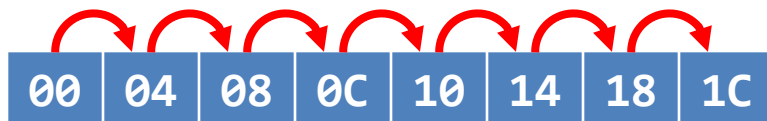
Cache Design Parameter 8: Prefetching

Prefetching

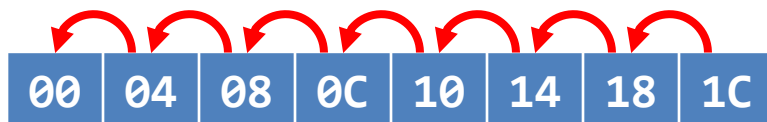
- **Prefetching**: fetching data that is expected to be needed soon
 - Allows you to hide the latency of fetching that data
 - E.g. Web browsers prefetch resources from not-yet-clicked links
 - when user later clicks on link, response is almost instantaneous
 - Caches also prefetch data that is expected to be used soon
 - Can be used to avoid even **cold misses**
- Two ways prefetching can happen:
 - Compiler-driven: compiler emits **prefetch instructions**
 - Can manually insert one in C program: `__builtin_prefetch(addr)`
 - Or rely on compiler to insert them using heuristics
 - Hardware-driven: CPU emits prefetches dynamically
 - Relies on CPU to detect a pattern in memory accesses

Hardware Prefetching

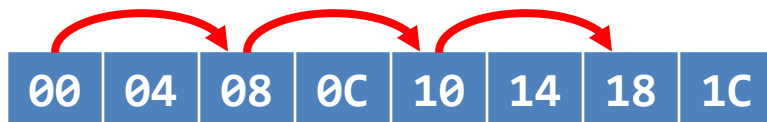
- What do you notice about both these snippets of code?
- They both access memory **sequentially**. **for**(i = 0 .. 100000)
 ○ The first one data, the next instructions. **A[i]++;**
- These kinds of access patterns are very common.



Sequential



Reverse sequential

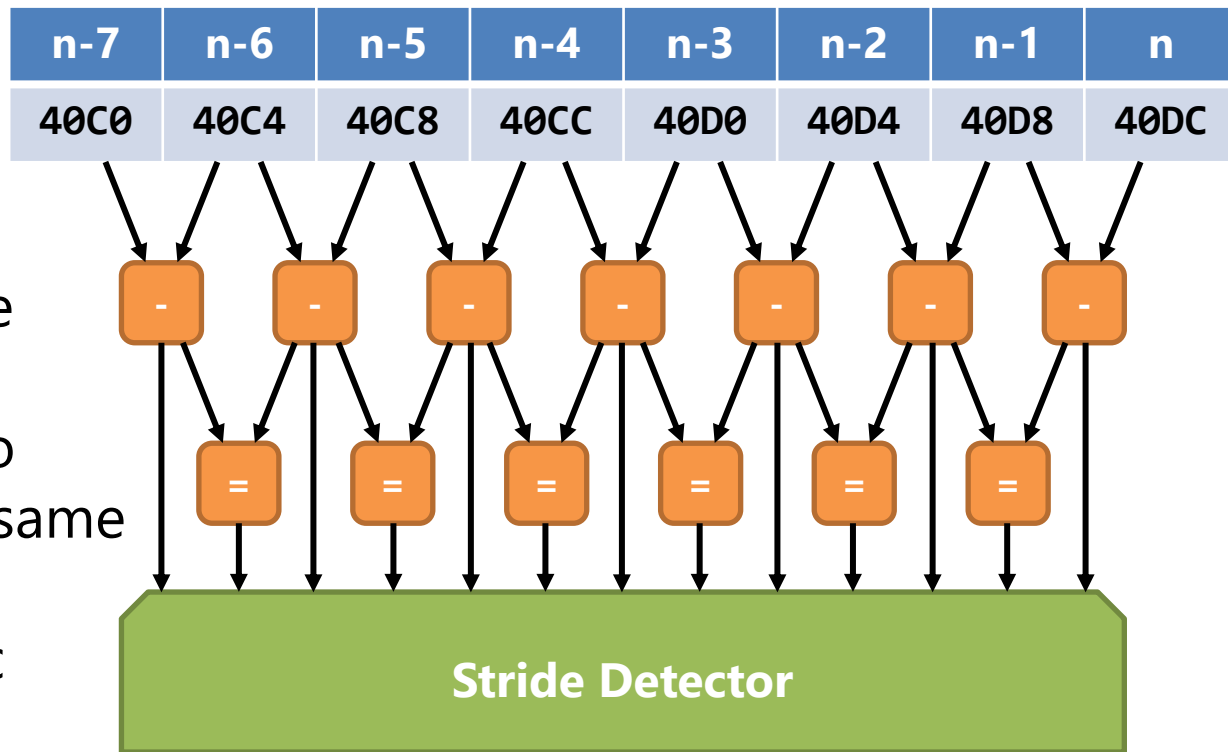


Strided sequential
(think "accessing one field
from each item in an array
of structs")

```
00 lw
04 lw
08 lw
0C addi
10 sub
14 mul
18 sw
1C sw
20 sw
```

Hardware Prefetching Stride Detection

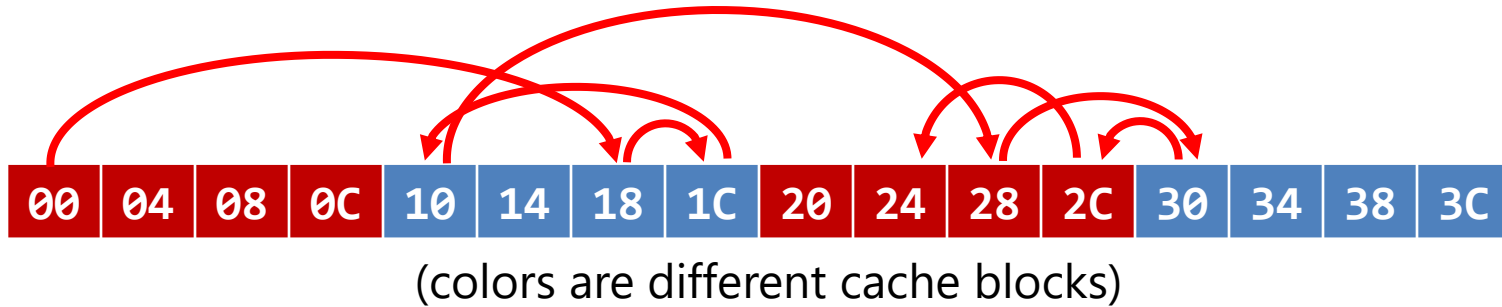
- What kinds of things would you need?
- A table of the last n memory accesses would be a good start.



- Some subtractors to calculate the stride
- Some comparators to see if strides are the same
- Some detection logic

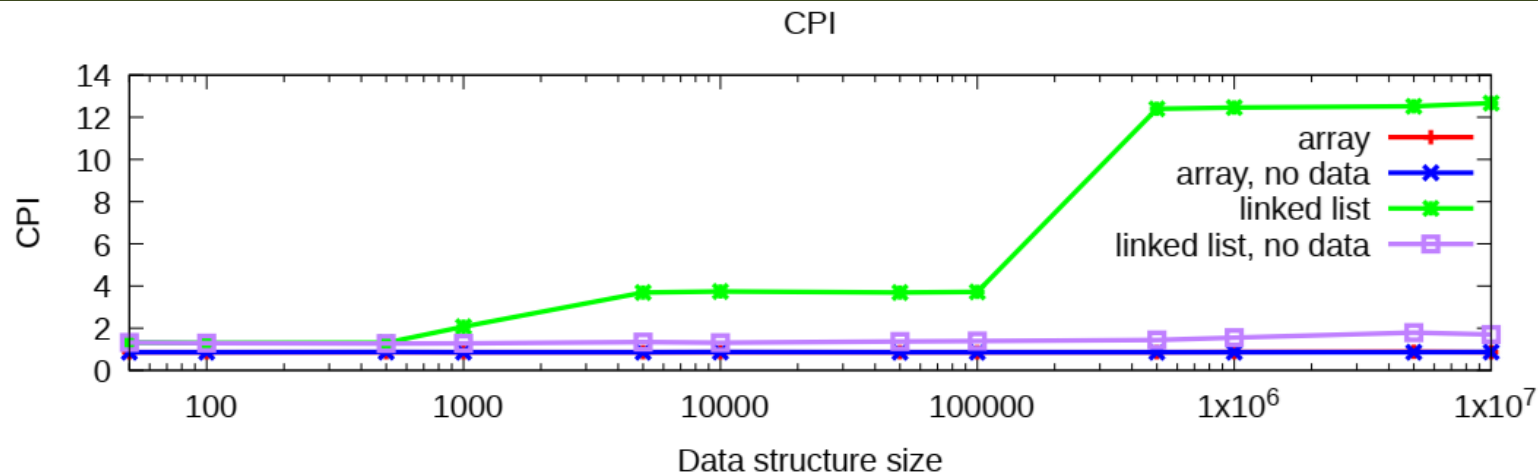
Where Hardware Prefetching Doesn't Work

- Sequential accesses are where prefetcher works best
 - E.g. Iterating over elements of an **array**
- Some accesses don't have a pattern or is too complex to detect
 - At below is how a typical **linked-list** traversal looks like



- Other pointer-chasing data structures (graphs, trees) look similar
- Can only rely on spatial locality to avoid cold misses

Mystery Solved



- How come **Array** performed well for even an array 1.28 GB large?
 - No spatial locality since each node takes up two 64-byte cache blocks
 - No temporal locality since working set of 1.28 GB exceeds any cache
- The answer is: **Array** had the benefit of a strided **prefetcher**!
 - Access pattern of **Linked List** was too complex for prefetcher to detect

Impact of Prefetching

- Prefetcher runs **in parallel** with the rest of the cache hardware
 - Does not slow down any on-demand reads or writes
- What if prefetcher is wrong? It can be wrong in two ways:
 - It fetched a block that was never going to be used
 - It fetched a useful block but fetched it too soon or too late
 - Too soon: the block gets evicted before it can be used
 - Too late: the prefetch doesn't happen in time for the access
- A bad prefetch results in **cache pollution**
 - Unused data is fetched, potentially pushing out other useful data
- On the other hand, good prefetches can reduce misses drastically!