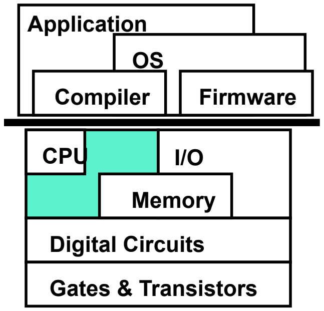
CIS 501
Computer Architecture

Unit 5: Caches

Slides originally developed by Amir Roth with contributions by Milo Martin at University of Pennsylvania with sources that included University of Wisconsin slides by Mark Hill, Guri Sohi, Jim Smith, and David Wood.

This Unit: Caches



- Basic memory hierarchy concepts
  - Speed vs capacity
- Caches
- Advanced memory hierarchy
- Later
  - Virtual memory
- Note: basic caching should be review, but some new stuff

Readings

- MA:FSPTCM
  - Section 2.2
  - Sections 6.1, 6.2, 6.3.1
- Paper:
  - Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers", ISCA 1990
    - ISCA's "most influential paper award" awarded 15 years later

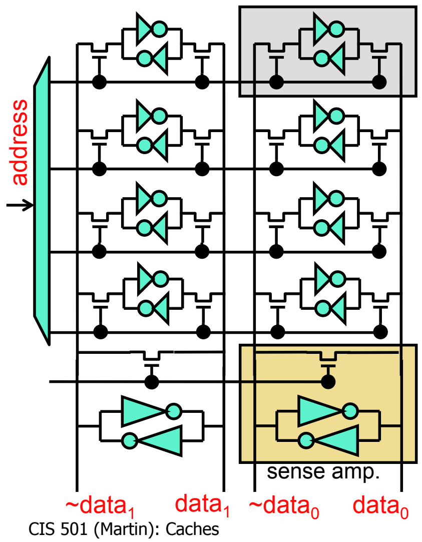
Motivation

- Processor can compute only as fast as memory
  - A 3Ghz processor can execute an "add" operation in 0.33ns
  - Today's "Main memory" latency is more than 33ns
  - Naïve implementation: loads/stores can be 100x slower than other operations
- Unobtainable goal:
  - Memory that operates at processor speeds
  - Memory as large as needed for all running programs
  - Memory that is cost effective
- Can't achieve all of these goals at once

Types of Memory

- **Static RAM (SRAM)**
  - 6 0r 8 transistors per bit
    - Two inverters (4 transistors) + transistors for reading/writing
  - Optimized for speed (first) and density (second)
  - Fast (sub-nanosecond latencies for small SRAM)
    - Speed roughly proportional to its area
  - Mixes well with standard processor logic
- **Dynamic RAM (DRAM)**
  - 1 transistor + 1 capacitor per bit
  - Optimized for density (in terms of cost per bit)
  - Slow (>40ns internal access, ~100ns pin-to-pin)
  - Different fabrication steps (does not mix well with logic)
- Nonvolatile storage: Magnetic disk, Flash RAM

SRAM Circuit Implementation

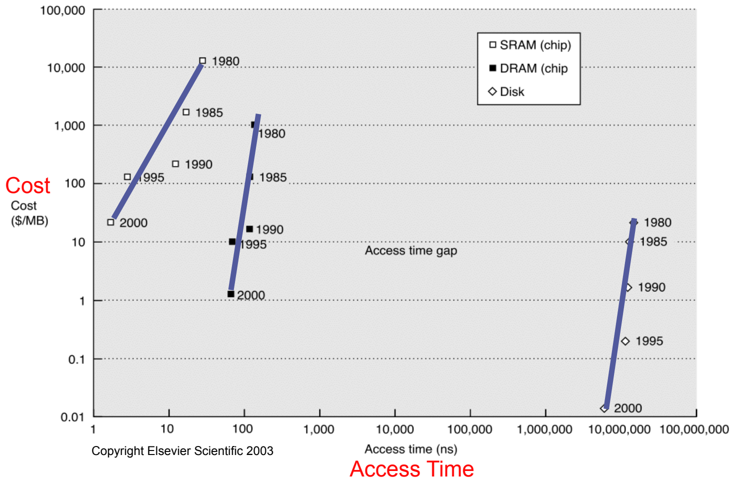


- SRAM:
  - Six transistors (6T) cells
  - 4 for the cross-coupled inverters
  - 2 access transistors
- **"Static"**
  - Cross-coupled inverters hold state
- To read
  - Equalize, swing, amplify
- To write
  - Overwhelm

Memory & Storage Technologies

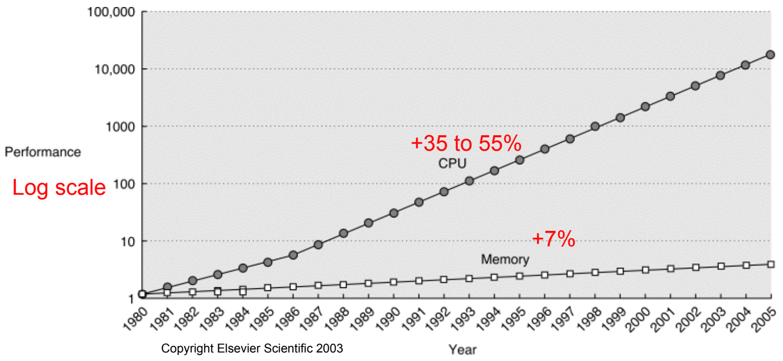
- **Cost** - what can $200 buy today (2009)?
  - SRAM: 16MB
  - DRAM: 4,000MB (4GB) – 250x cheaper than SRAM
  - Flash: 64,000MB (64GB) – 16x cheaper than DRAM
  - Disk: 2,000,000MB (2TB) – 32x vs. Flash (512x vs. DRAM)
- **Latency**
  - SRAM: <1 to 2ns (on chip)
  - DRAM: ~50ns – 100x or more slower than SRAM
  - Flash: 75,000ns (75 microseconds) – 1500x vs. DRAM
  - Disk: 10,000,000ns (10ms) – 133x vs Flash (200,000x vs DRAM)
- **Bandwidth**
  - SRAM: 300GB/sec (e.g., 12-port 8-byte register file @ 3Ghz)
  - DRAM: ~25GB/s
  - Flash: 0.25GB/s (250MB/s), 100x less than DRAM
  - Disk: 0.1 GB/s (100MB/s), 250x vs DRAM, **sequential** access only

Memory Technology Trends



Copyright Elsevier Scientific 2003

The "Memory Wall"



Copyright Elsevier Scientific 2003

- Processors are get faster more quickly than memory (note log scale)
  - Processor speed improvement: 35% to 55%
  - Memory latency improvement: 7%

Known From the Beginning

"Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available … We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible."

Burks, Goldstine, VonNeumann
"Preliminary discussion of the logical design of an electronic computing instrument"
IAS memo 1946

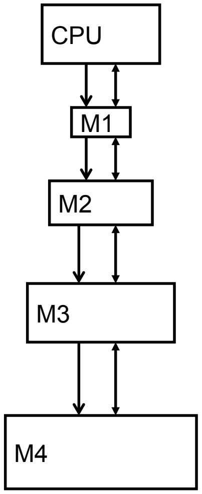
Locality to the Rescue

- **Locality of memory references**
  - Property of real programs, few exceptions
  - Books and library analogy (next slide)
- **Temporal locality**
  - Recently referenced data is likely to be referenced again soon
  - **Reactive**: cache recently used data in small, fast memory
- **Spatial locality**
  - More likely to reference data near recently referenced data
  - **Proactive**: fetch data in large chunks to include nearby data
- Holds for data and instructions

Library Analogy

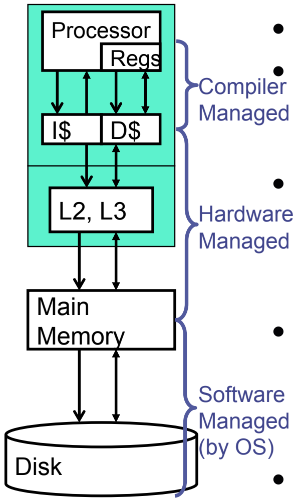
- Consider books in a library
- Library has lots of books, but it is slow to access
  - Far away (time to walk to the library)
  - Big (time to walk within the library)
- How can you avoid these latencies?
  - Check out books, take them home with you
    - Put them on desk, on bookshelf, etc.
  - But desks & bookshelves have limited capacity
    - Keep recently used books around (**temporal locality**)
    - Grab books on related topic at the same time (**spatial locality**)
    - Guess what books you'll need in the future (prefetching)

Exploiting Locality: Memory Hierarchy



- Hierarchy of memory components
  - Upper components
    - Fast ⇔ Small ⇔ Expensive
  - Lower components
    - Slow ⇔ Big ⇔ Cheap
- Connected by "buses"
  - Which also have latency and bandwidth issues
- Most frequently accessed data in M1
  - M1 + next most frequently accessed in M2, etc.
  - Move data up-down hierarchy
- Optimize average access time
  - ***$latency_{avg}$ = $latency_{hit}$ + $\%_{miss}$ * $latency_{miss}$***
  - Attack each component

Concrete Memory Hierarchy

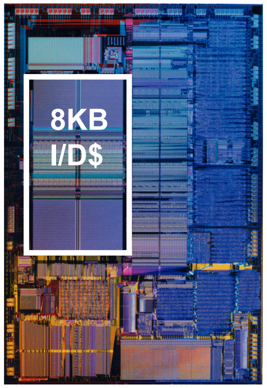


- 0th level: **Registers**
- 1st level: **Primary caches**
  - Split instruction (I$) and data (D$)
  - Typically 8KB to 64KB each
- 2nd level: **2nd and 3rd cache** (L2, L3)
  - On-chip, typically made of SRAM
  - 2nd level typically ~256KB to 512KB
  - "Last level cache" typically 4MB to 16MB
- 3rd level: **main memory**
  - Made of DRAM ("Dynamic" RAM)
  - Typically 1GB to 4GB for desktops/laptops
    - Servers can have 100s of GB
- 4th level: **disk (swap and files)**
  - Uses magnetic disks

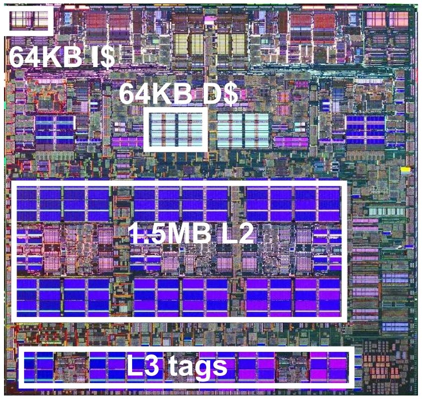
Library Analogy Revisited

- Registers ⇔ books on your desk
  - Actively being used, small capacity
- Caches ⇔ bookshelves
  - Moderate capacity, pretty fast to access
- Main memory ⇔ library
  - Big, holds almost all data, but slow
- Disk (swap) ⇔ inter-library loan
  - Very slow, but hopefully really uncommon

Evolution of Cache Hierarchies



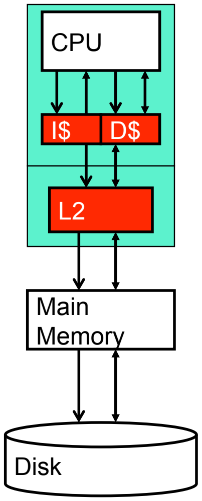
Intel 486



IBM Power5 (dual core)

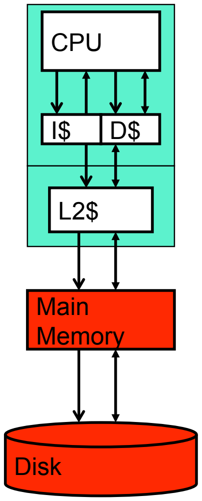
- Chips today are 30–70% cache by area

This Unit: Caches



- "Cache": hardware managed
  - Hardware automatically retrieves missing data
  - Built from fast SRAM, usually on-chip today
  - In contrast to off-chip, DRAM "main memory"
- Cache organization
  - ABC
  - Miss classification
- High-performance techniques
  - Reducing misses
  - Improving miss penalty
  - Improving hit latency
- Some example performance calculations

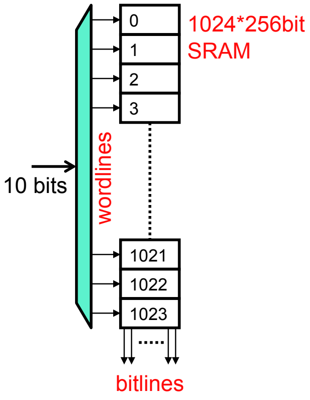
Memory and Disk



- Main memory
  - DRAM-based memory systems
  - Virtual memory
- Disks and Storage
  - Disks vs Flash
  - Disk arrays (for performance and reliability)
- Likely not covering disks this year
  - Make room for more on multicore

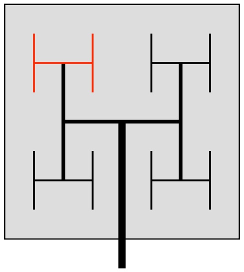
Basic Memory Array Structure

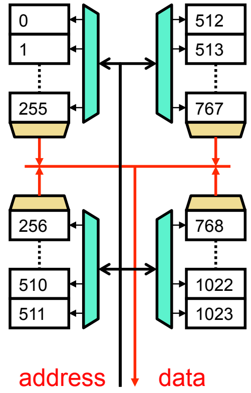
- Number of entries
  - 2^n, where n is number of address bits
  - Example: 1024 entries, 10 bit address
  - Decoder changes n-bit address to 2^n bit "one-hot" signal
  - One-bit address travels on "wordlines"
- Size of entries
  - Width of data accessed
  - Data travels on "bitlines"
  - 256 bits (32 bytes) in example



FYI: Physical Memory Layout

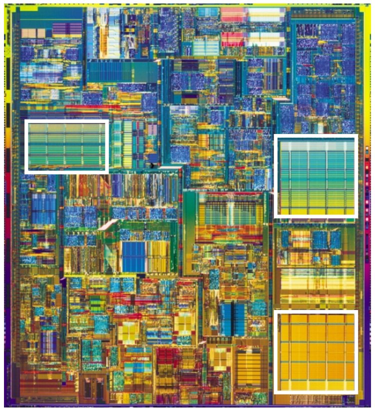
- Logical layout
  - Arrays are vertically contiguous
- Physical layout - roughly square
  - Vertical partitioning to minimize wire lengths
  - **H-tree**: horizontal/vertical partitioning layout
    - Applied recursively
    - Each node looks like an H





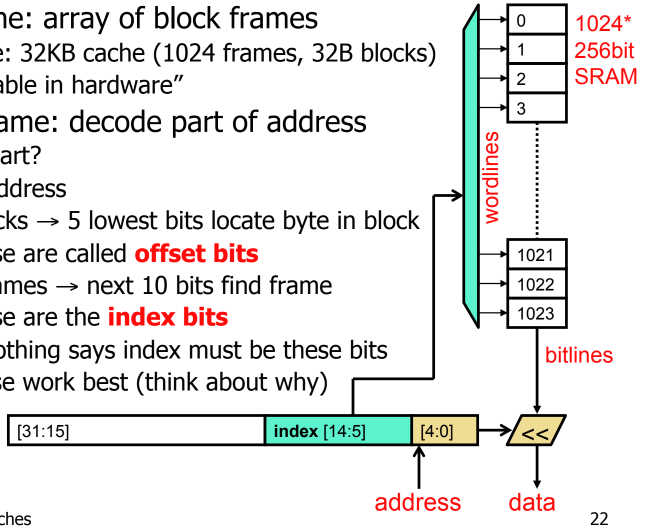
Physical Cache Layout

- Arrays and h-trees make caches easy to spot in μgraphs



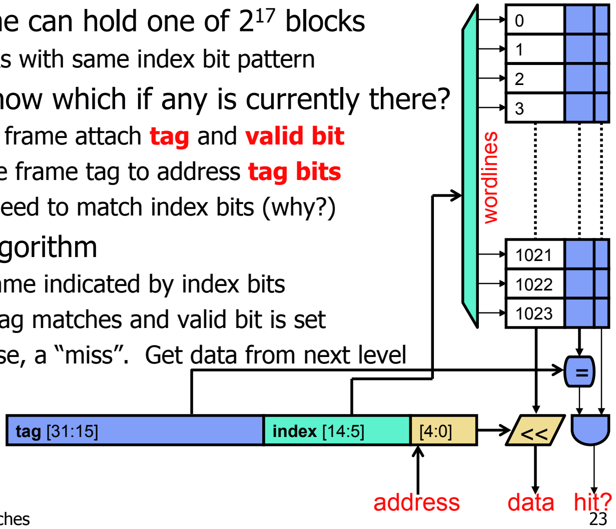
Caches: Finding Data via Indexing

- Basic cache: array of block frames
  - Example: 32KB cache (1024 frames, 32B blocks)
  - “Hash table in hardware”
- To find frame: decode part of address
  - Which part?
  - 32-bit address
  - 32B blocks → 5 lowest bits locate byte in block
    - These are called **offset bits**
  - 1024 frames → next 10 bits find frame
    - These are the **index bits**
  - Note: nothing says index must be these bits
  - But these work best (think about why)



Knowing that You Found It: Tags

- Each frame 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 frame attach **tag** and **valid bit**
  - Compare frame tag to address **tag bits**
    - No need to match index bits (why?)
- Lookup algorithm
  - Read frame indicated by index bits
  - “Hit” if tag matches and valid bit is set
  - Otherwise, a “miss”. Get data from next level



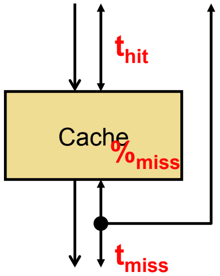
Calculating Tag Overhead

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

Handling a Cache Miss

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

Cache Performance Equation



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

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

CPI Calculation with Cache Misses

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

Measuring Cache Performance

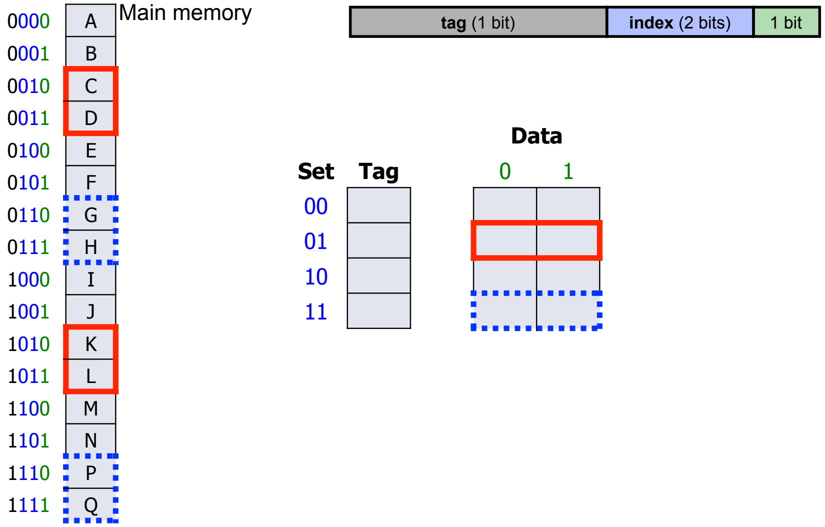
- Ultimate metric is t_{avg}
  - Cache capacity and circuits roughly determines t_{hit}
  - Lower-level memory structures determine t_{miss}
  - Measure $\%_{miss}$
    - Hardware performance counters
    - Simulation (homework)
    - Paper simulation (next)

Cache Examples

- 4-bit addresses → 16B memory
  - Simpler cache diagrams than 32-bits
- 8B cache, 2B blocks
  - Figure out number of sets: 4 (capacity / block-size)
  - Figure out how address splits into offset/index/tag bits
    - Offset: least-significant $\log_2$(block-size) = $\log_2$(2) = 1 → 0000
    - Index: next $\log_2$(number-of-sets) = $\log_2$(4) = 2 → 0000
    - Tag: rest = 4 – 1 – 2 = 1 → 0000

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

4-bit Address, 8B Cache, 2B Blocks

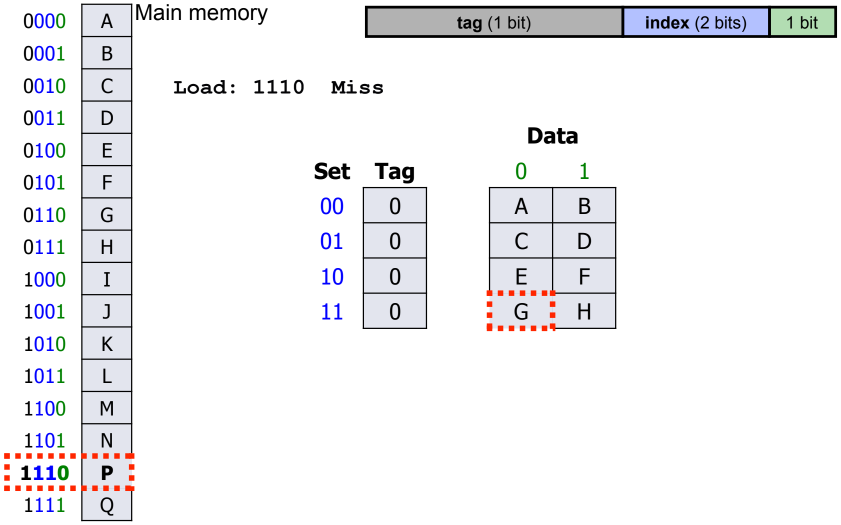


	Main memory
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	I
1001	J
1010	K
1011	L
1100	M
1101	N
1110	P
1111	Q

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

Set	Tag	Data 0	Data 1
00			
01			
10			
11			

4-bit Address, 8B Cache, 2B Blocks



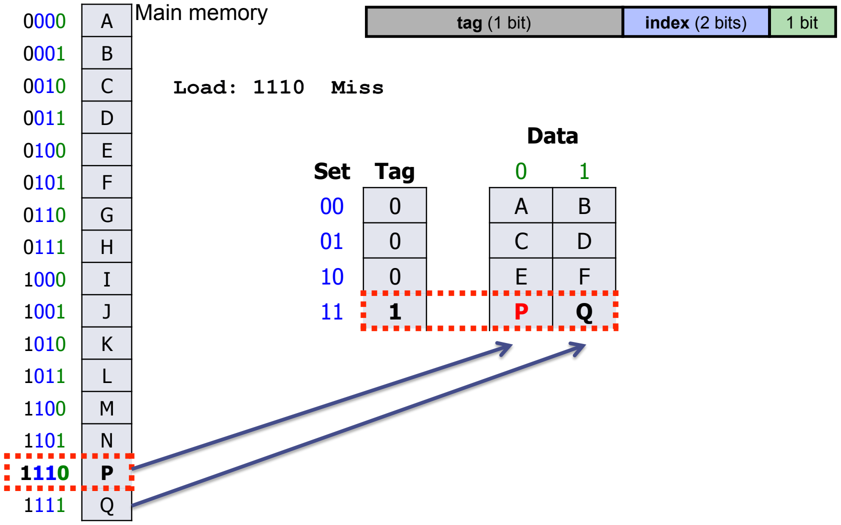
	Main memory
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	I
1001	J
1010	K
1011	L
1100	M
1101	N
1110	**P**
1111	Q

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

```
Load: 1110  Miss
```

Set	Tag	Data 0	Data 1
00	0	A	B
01	0	C	D
10	0	E	F
11	0	G	H

4-bit Address, 8B Cache, 2B Blocks



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

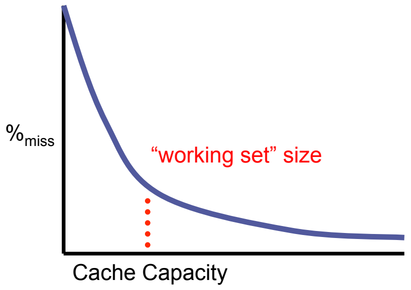
tag (1 bit)	index (2 bits)	1 bit

```
Load: 1110  Miss
```

Set	Tag	Data 0	Data 1
00	0	A	B
01	0	C	D
10	0	E	F
11	**1**	**P**	**Q**

Capacity and Performance

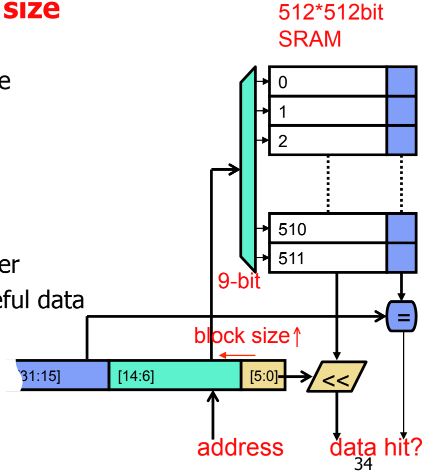
- Simplest way to reduce $\%_{miss}$: increase capacity
  - + Miss rate decreases monotonically
    - **"Working set"**: insns/data program is actively using
    - Diminishing returns
  - – However t_{hit} increases
    - Latency proportional to sqrt(capacity)
  - t_{avg} ?



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

Block Size

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



Block Size and Tag Overhead

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

4-bit Address, 8B Cache, 4B Blocks

Main memory

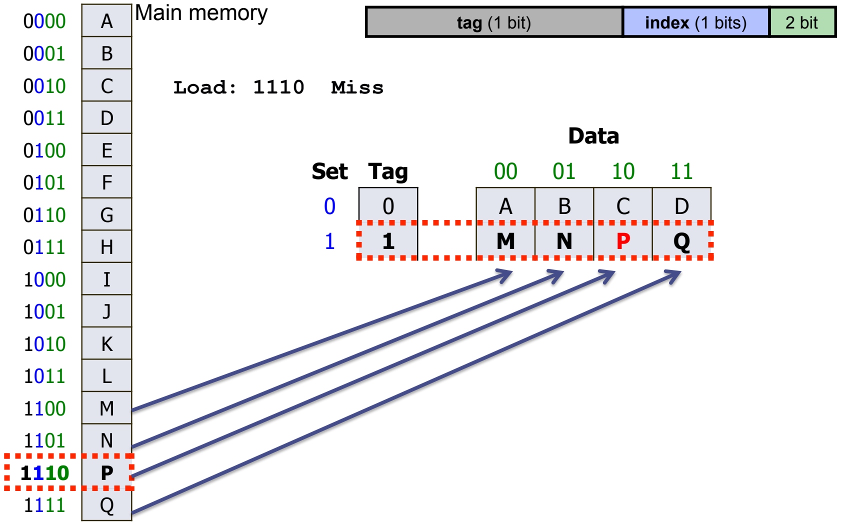
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	I
1001	J
1010	K
1011	L
1100	M
1101	N
1110	**P**
1111	Q

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

`Load: 1110  Miss`

		Data			
Set	Tag	00	01	10	11
0	0	A	B	C	D
1	0	E	F	G	H

4-bit Address, 8B Cache, 4B Blocks



Main memory

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

Load: 1110 Miss

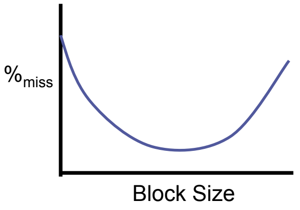
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	I
1001	J
1010	K
1011	L
1100	M
1101	N
1110	**P**
1111	Q

Data

Set	Tag	00	01	10	11
0	0	A	B	C	D
1	**1**	**M**	**N**	**P**	**Q**

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 frames)
    - Turns hits into misses by disallowing simultaneous residence
    - Consider entire cache as one big block

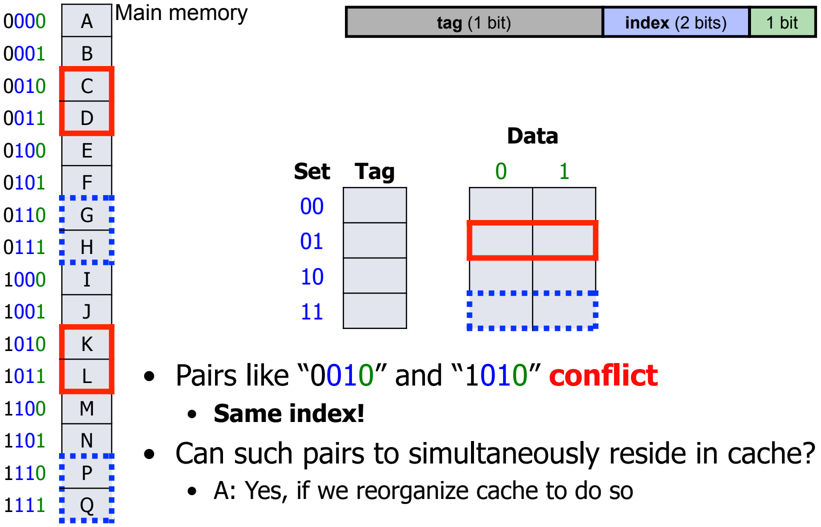


- Both effects always present
  - Spatial prefetching dominates initially
    - Depends on size of the cache
  - Good block size is 16–128B
    - Program dependent

Block Size and Miss Penalty

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

Cache Conflicts



Main memory

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

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

Data

Set	Tag	0	1
00			
01			
10			
11			

- Pairs like “0010” and “1010” **conflict**
  - **Same index!**
- Can such pairs to simultaneously reside in cache?
  - A: Yes, if we reorganize cache to do so

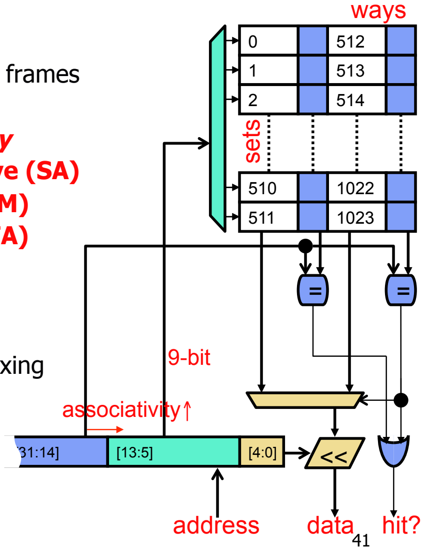
Set-Associativity

- **Set-associativity**
  - Block can reside in one of few frames
  - Frame groups called ***sets***
  - Each frame 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$_{\text{hit:}}$
  - additional tag match & muxing

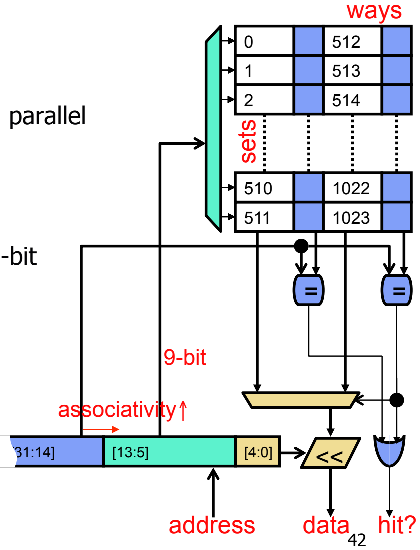
  - Note: valid bit not shown



Set-Associativity

- Lookup algorithm
  - Use index bits to find set
  - Read data/tags in all frames 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



Replacement Policies

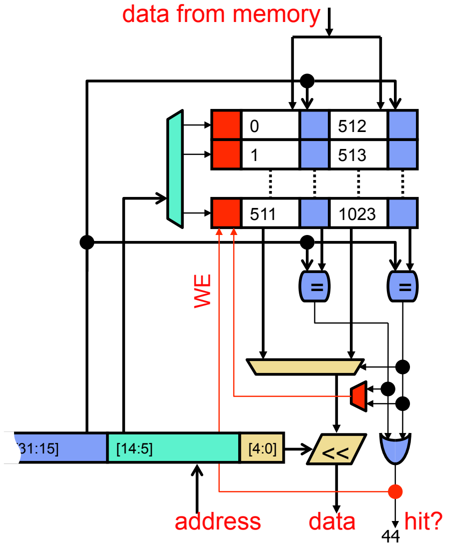
- Set-associative caches present a new design choice
  - On cache miss, which block in set to replace (kick out)?
- Some options
  - **Random**
  - **FIFO (first-in first-out)**
  - **LRU (least recently used)**
    - Fits with temporal locality, LRU = least likely to be used in future
  - **NMRU (not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

  - Which policy is simulated in previous example?

NMRU and Miss Handling

- Add **LRU** field to each set
  - "Least recently used"
  - LRU data is encoded "way"
  - Hit? update MRU

- LRU bits updated on each access



4-bit Address, 8B Cache, 2B Blocks, 2-way

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

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

	Way 0			LRU	Way 1		
		Data				Data	
Set	Tag	0	1		Tag	0	1
0							
1							

4-bit Address, 8B Cache, 2B Blocks, 2-way

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

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

`Load: 1110 Miss`

	Way 0			LRU	Way 1		
		Data				Data	
Set	Tag	0	1		Tag	0	1
0	00	A	B	0	01	E	F
1	00	C	D	1	01	G	H

4-bit Address, 8B Cache, 2B Blocks, 2-way

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

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

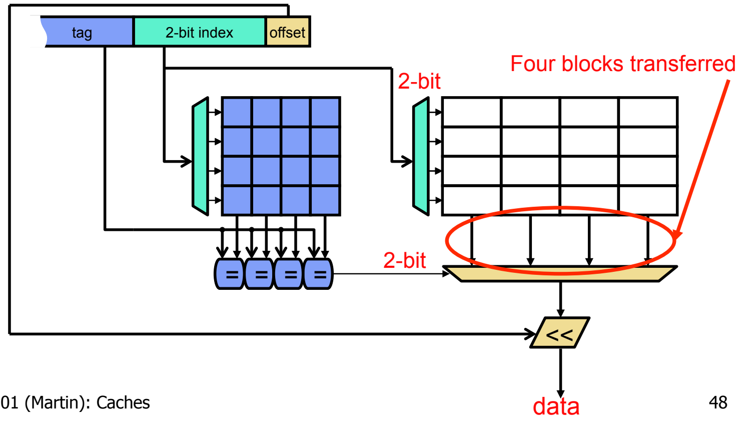
`Load: 1110 Miss`

	Way 0			LRU	Way 1		
		Data				Data	
Set	Tag	0	1		Tag	0	1
0	00	A	B	0	01	E	F
1	00	C	D	**0**	**11**	**P**	**Q**

LRU updated on each access (not just misses)

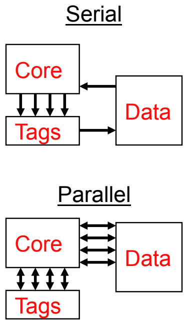
Parallel or Serial Tag Access?

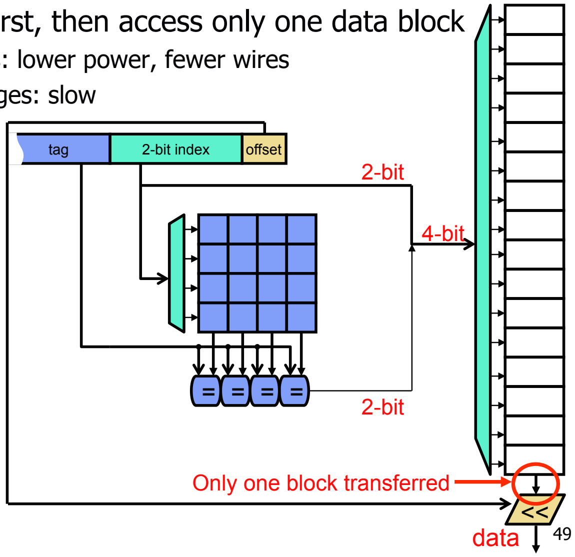
- Note: data and tags actually physically separate
  - Split into two different arrays
- Parallel access example:



Serial Tag Access

- Tag match first, then access only one data block
  - Advantages: lower power, fewer wires
  - Disadvantages: slow

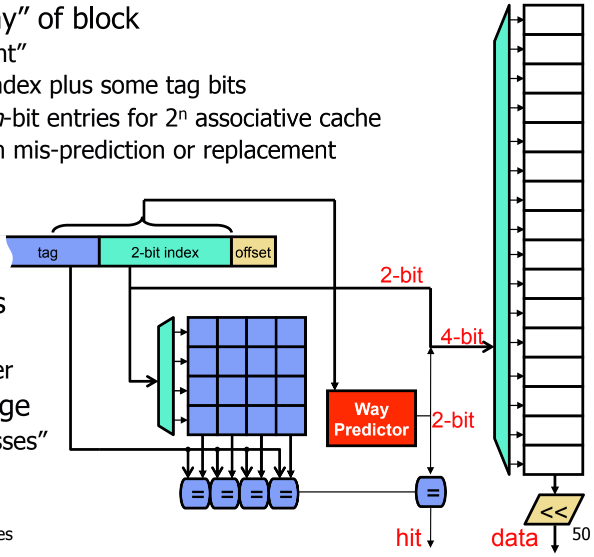




Only one block transferred

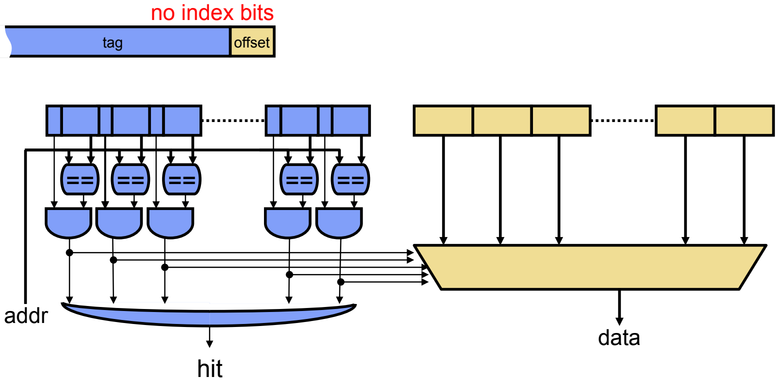
Best of Both? Way Prediction

- Predict "way" of block
  - Just a "hint"
  - Use the index plus some tag bits
  - Table of *n*-bit entries for 2^n associative cache
  - Update on mis-prediction or replacement
- Advantages
  - Fast
  - Low-power
- Disadvantage
  - More "misses"



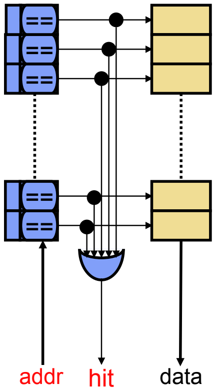
Full Associativity

- How to implement full (or at least high) associativity?
  - This way is terribly inefficient
  - 1K matches are unavoidable, but 1K data reads + 1K-to-1 mux?



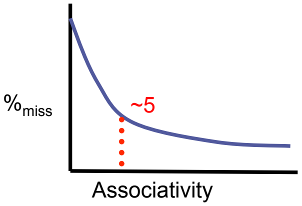
Full Associativity with CAMs

- **CAM**: content addressable memory
  - Array of words with built-in comparators
  - Input is data (tag)
  - Output is 1-hot encoding of matching slot
- Fully associative cache
  - Tags as CAM, data as RAM
  - Effective but somewhat expensive
    - But cheaper than any other way
  - Upshot: used for 16-/32-way associativity
  - – No good way to build 1024-way associativity
  - + No real need for it, either
- CAMs are used elsewhere, too



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

Classifying Misses: 3C Model (Hill)

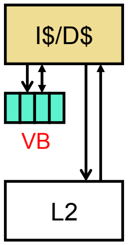
- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - **Would miss even in infinite cache**
  - **Capacity**: miss caused because cache is too small
    - **Would miss even in fully associative cache**
    - Identify? Consecutive accesses to block separated by access to at least N other distinct blocks (N is number of frames 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

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 frames)
  - + Decreases compulsory/capacity misses (spatial locality)
  - No significant effect on latency_{hit}
- **Capacity**
  - + Decreases capacity misses
  - – Increases latency_{hit}

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}**?



Overlapping Misses: Lockup Free Cache

- **Lockup free**: allows other accesses while miss is pending
  - Consider: Load [r1] -> r2;   Load [r3] -> r4;   Add r2, r4 -> r5
  - **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 frame, 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

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];
```

  - Better code

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

Software Restructuring: Data

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

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

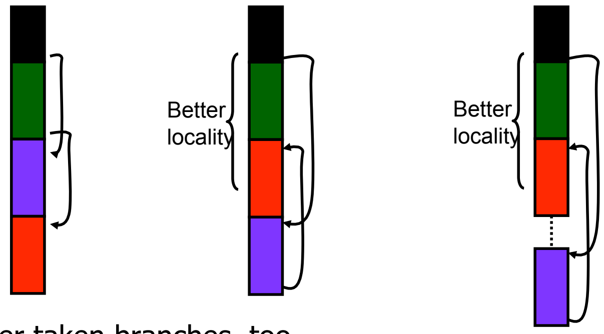
  - Better code
    - Cut array into CACHE_SIZE chunks
    - Run all phases on one chunk, proceed to next chunk

```
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

Software Restructuring: Code

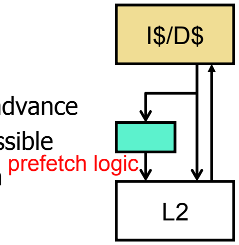
- Compiler an layout code for temporal and spatial locality
  - If (a) { **code1;** } else { **code2;** } **code3;**
  - But, code2 case never happens (say, error condition)



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

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 address **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



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
  - More "Memory-level" parallelism (MLP)

Hardware Prefetching

- What to prefetch?
  - Use a hardware table to detect strides, common patterns
  - **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.
    - Large table records (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

Write Issues

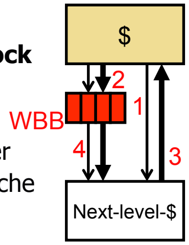
- So far we have looked at reading from cache
  - Instruction fetches, loads
- What about writing into cache
  - Stores, not an issue for instruction caches (why they are simpler)
- Several new issues
  - Tag/data access
  - Write-through vs. write-back
  - Write-allocate vs. write-not-allocate
  - Hiding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

Tag/Data Access

- Reads: read tag and data in parallel
  - Tag mis-match → data is wrong (OK, just stall until good data arrives)
- Writes: read tag, write data in parallel? No. Why?
  - 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

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
    - Step#1: Send "fill" request to next-level
    - Step#2: While waiting, write dirty block to buffer
    - Step#3: When new blocks arrives, put it into cache
    - Step#4: Write buffer contents to next-level



Write Propagation Comparison

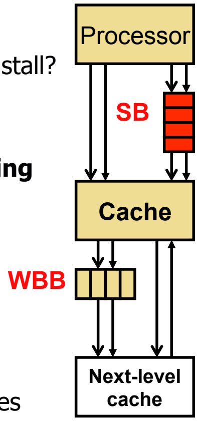
- **Write-through**
  - – Requires additional bus bandwidth
    - Consider repeated write hits
  - – Next level must handle small writes (1, 2, 4, 8-bytes)
  - + No need for dirty bits in cache
  - + No need to handle "writeback" operations
    - Simplifies miss handling (no write-back buffer)
  - Sometimes 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
  - Second-level and beyond are generally write-back caches

Write Miss Handling

- How is a write miss actually 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

Write Misses and Store Buffers

- Read miss?
  - Load can't go on without the data, it must stall
- Write miss?
  - Technically, no instruction is waiting for data, why stall?
- **Store buffer**: a small buffer
  - Stores put address/value to store buffer, **keep going**
  - Store buffer writes stores to D$ in the background
  - Loads must search store buffer (in addition to D$)
  - + Eliminates stalls on write misses (mostly)
  - – Creates some problems (later)
- Store buffer vs. writeback-buffer
  - Store buffer: "in front" of D$, for hiding store misses
  - Writeback buffer: "behind" D$, for hiding writebacks



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 $\%_{miss}$)

Memory Hierarchy Parameters

Parameter	**I$/D$**	**L2**	**L3**	**Main Memory**
t_{hit}	2ns	10ns	30ns	100ns
t_{miss}	**10ns**	**30ns**	**100ns**	**10ms (10M ns)**
Capacity	8KB–64KB	256KB–8MB	2–16MB	1-4GBs
Block size	16B–64B	32B–128B	32B-256B	NA
Associativity	1–4	4–16	4-16	NA

- Some other design parameters
  - Split vs. unified insns/data
  - Inclusion vs. exclusion vs. nothing

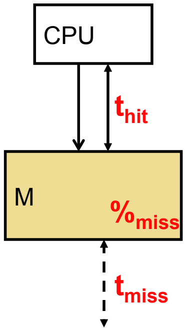
Split vs. Unified Caches

- **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
  - Why is 486 I/D$ unified?
- **Unified L2, L3**: insns and data together
  - To minimize $\%_{miss}$
  - + Fewer capacity misses: unused insn capacity can be 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

Hierarchy: Inclusion versus Exclusion

- **Inclusion**
  - Bring block from memory into L2 then L1
    - A block in the L1 is always in the L2
  - If block evicted from L2, must also evict it from L1
    - More on this when we talk about multicore
- **Exclusion**
  - Bring block from memory into L1 but not (L2 or lower)
    - L2 becomes a large victim cache
    - 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

Memory Performance Equation



- For memory component M
  - **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

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

Performance Calculation

- Parameters
  - Base pipeline CPI = 1
    - In this case, already incorporates t_{hit}
  - Instruction mix: 30% loads/stores
  - I$: $\%_{miss}$ = 2% of accesses, t_{miss} = 10 cycles
  - D$: $\%_{miss}$ = 10% of accesses, t_{miss} = 10 cycles

- What is new CPI?
  - $CPI_{I\$} = \%_{missI\$} * t_{miss}$ = 0.02*10 cycles = 0.2 cycle
  - $CPI_{D\$} = \%_{memory} * \%_{missD\$} * t_{missD\$}$ = 0.30*0.10*10 cycles = 0.3 cycle
  - $CPI_{new} = CPI + CPI_{I\$} + CPI_{D\$}$ = 1+0.2+0.3= 1.5

Performance Calculation (Revisited)

- Parameters
  - Base pipeline CPI = 1
    - In this case, already incorporates t_{hit}

  - I$: $\%_{miss}$ = 2% of instructions, t_{miss} = 10 cycles
  - D$: $\%_{miss}$ = 3% of instructions, t_{miss} = 10 cycles

- What is new CPI?
  - $CPI_{I\$} = \%_{missI\$} * t_{miss}$ = 0.02*10 cycles = 0.2 cycle
  - $CPI_{D\$} = \%_{missD\$} * t_{missD\$}$ = 0.03*10 cycles = 0.3 cycle
  - $CPI_{new} = CPI + CPI_{I\$} + CPI_{D\$}$ = 1+0.2+0.3= 1.5

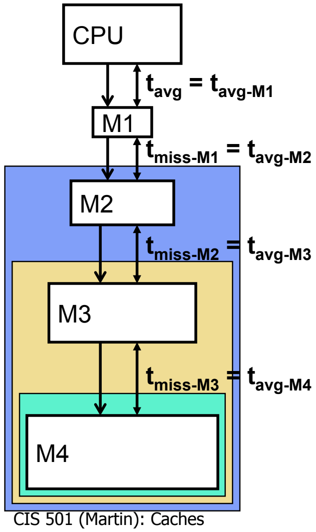
Miss Rates: per "access" vs "instruction"

- Miss rates can be expressed two ways:
  - Misses per "instruction" (or instructions per miss), -or-
  - Misses per "cache access" (or accesses per miss)

- For first-level caches, use instruction mix to convert
  - If memory ops are 1/3rd of instructions..
  - 2% of instructions miss (1 in 50) is 6% of "accesses" miss (1 in 17)

- What about second-level caches?
  - Misses per "instruction" still straight-forward ("global" miss rate)
  - Misses per "access" is trickier ("local" miss rate)
    - Depends on number of accesses (which depends on L1 rate)

Hierarchy Performance


CPU
$t_{avg} = t_{avg\text{-}M1}$
M1
$t_{miss\text{-}M1} = t_{avg\text{-}M2}$
M2
$t_{miss\text{-}M2} = t_{avg\text{-}M3}$
M3
$t_{miss\text{-}M3} = t_{avg\text{-}M4}$
M4

t_{avg}

$t_{avg\text{-}M1}$

$t_{hit\text{-}M1} + (\%_{miss\text{-}M1} * t_{miss\text{-}M1})$

$t_{hit\text{-}M1} + (\%_{miss\text{-}M1} * t_{avg\text{-}M2})$

$t_{hit\text{-}M1} + (\%_{miss\text{-}M1} * (t_{hit\text{-}M2} + (\%_{miss\text{-}M2} * t_{miss\text{-}M2})))$

$t_{hit\text{-}M1} + (\%_{miss\text{-}M1} * (t_{hit\text{-}M2} + (\%_{miss\text{-}M2} * t_{avg\text{-}M3})))$

…

Multilevel Performance Calculation II

- Parameters
  - 30% of instructions are memory operations
  - L1: t_{hit} = 1 cycles (included in CPI of 1), $\%_{miss}$ = 5% of accesses
  - L2: t_{hit} = 10 cycles, $\%_{miss}$ = 20% of L2 accesses
  - Main memory: t_{hit} = 50 cycles
- Calculate CPI
  - CPI = 1 + 30% * 5% * $t_{missD\$}$
  - $t_{missD\$} = t_{avgL2} = t_{hitL2} + \%_{missL2} * t_{hitMem}$ = 10 + (20% * 50) = 20 cycles
  - Thus, CPI = 1 + 30% * 5% * 20 = 1.3 CPI
- Alternate CPI calculation:
  - What % of instructions miss in L1 cache? 30%*5% = 1.5%
  - What % of instructions miss in L2 cache? 20%*1.5% = 0.3% of insn
  - CPI = 1 + (1.5% * 10) + (0.3% * 50) = 1 + 0.15 + 0.15 = 1.3 CPI

Summary

- **Average access time** of a memory component
  - $latency_{avg} = latency_{hit} + \%_{miss} * latency_{miss}$
  - Hard to get low $latency_{hit}$ and $\%_{miss}$ in one structure → hierarchy
- **Memory hierarchy**
  - Cache (SRAM) → memory (DRAM) → swap (Disk)
  - Smaller, faster, more expensive → bigger, slower, cheaper
- Cache ABCs (**capacity, associativity, block size**)
  - 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

Calculations: Book versus Lecture Notes

- My calculation equation:
  - latency_{avg} = latency_{hit} + $\%_{miss}$ * $\text{latency}_{miss\_additional}$
- The book uses a different equation:
  - latency_{avg} = (latency_{hit} * $\%_{hit}$) + ($\text{latency}_{miss\_total}$ * (1 - $\%_{hit}$))
- These are actually the same:
  - $\text{latency}_{miss\_total}$ = $\text{latency}_{miss\_additional}$ + latency_{hit}
  - $\%_{hit}$ = 1 - $\%_{miss}$, so: latency_{avg} =
  - **= (latency_{hit} * $\%_{hit}$) + ($\text{latency}_{miss\_total}$ * (1 - $\%_{hit}$))**
  - = (latency_{hit} * (1 - $\%_{miss}$)) + ($\text{latency}_{miss\_total}$ * $\%_{miss}$)
  - = latency_{hit} + $\text{latency}_{hit\,*}$ (- $\%_{miss}$) + ($\text{latency}_{miss\_total}$ * $\%_{miss}$)
  - = latency_{hit} + ($\%_{miss}$ * -1 * (latency_{hit} - $\text{latency}_{miss\_total}$))
  - = latency_{hit} + ($\%_{miss}$ * ($\text{latency}_{miss\_total}$ - latency_{hit}))
  - = latency_{hit} + ($\%_{miss}$ * ($\text{latency}_{miss\_total}$ - latency_{hit}))
  - **= latency_{hit} + ($\%_{miss}$ * $\text{latency}_{miss\_additional}$)**