ECE437: Introduction to Digital Computer Design

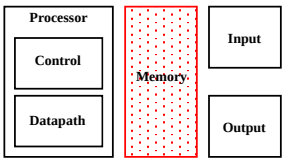
Chapter 5a (caches)

Fall 2016

Problem

- Why do we care about memory system?
  - CPU only as fast as mem-system can supply
  - You can pipeline till blue in the face but MANY stalls if memory is slow
  - Base CPI ~1.2, every 5th instr is ld, memory is 10 cycles → CPI ~3.2! (real memory is 300+ cycles → CPI ~61.2!!)
- Understand SRAM/DRAM technology
- Exploit program characteristics to overcome processor-memory gap

Outline



- Memory
  - Technology, organization, motivation for hierarchical organization

Memory

- Storage elements
  - registers, latches. — Processor Datapath
    - Small
    - In processor
    - Expensive to add (??)
  - SRAM (Caches)
    - Medium
    - Onchip, close to processor
    - Costly
  - DRAM (Main memory) — Memory subsystem
    - Large, Off chip, 50ns access time
    - Spring 2015, 0.42c/MB ($4.3/GB)
    - Spring 2010: 0.95c/MB (2GB for $19 *)
    - Spring 2008, $0.02/MB (4GB for $79 *)
    - In 2006 $0.06-0.10/MB (512MB for 35-45$ *)
  - Disk etc. — I/O subsystem
    - Large, far from processor, Slow (~ms)
    - Spring 2015, 3.3c/GB (3 TB for $99)
    - Spring 2010, 7.5c/GB (1TB for $75*)
    - Fall 2008, $0.11/GB (1TB for $108.99)
    - In Spring 2008, $0.22/GB (500GB for $112 *)
    - In 2006 $0.35-0.40/GB (200GB for 70$ *)

Memory Hierarchy Technology

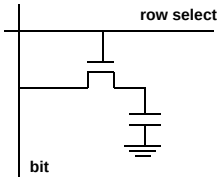
- Random Access:
  - "Random" is good: access time is the same for all locations
  - DRAM: Dynamic Random Access Memory
    - High density, low power, cheap, but slow
    - Dynamic: need to be "refreshed" regularly even if powered
  - SRAM: Static Random Access Memory
    - Fast but low density, high power, expensive
    - Static: content will last "forever" if powered
- "Not-so-random" Access Technology:
  - Access time varies from location to location and from time to time
  - Examples: Disk, CDROM
- Sequential Access Technology: access time linear in location (e.g.,Tape)
- Main Memory (DRAMs) + Caches (SRAMs)

DRAM

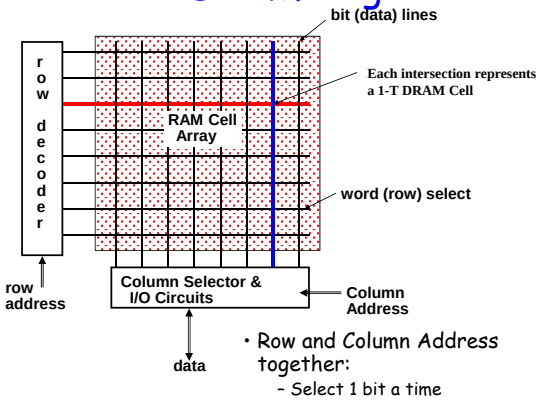
- Dynamic RAM
  - Dense, 1Transistor/bit-cell
  - Forgets after a while
  - 16Mb : 4K x 4K cell-array
    - 16Gb memory available now
  - 24 bit address
    - 12 bit for row, 12 for column—reflected in the interface
- Implementation
  - Word/byte DRAM built as DIMM/SIMMs

1T DRAM cell

- Charge on capacitor
- Write:
  - 1. Drive bit line
  - 2.. Select row
- Read:
  - 1. Precharge bit line to Vdd
  - 2.. Select row
  - 3. Cell and bit line share charges
    - Very small voltage changes on the bit line
  - 4. Sense (fancy sense amp)
    - Can detect changes of ~1 million electrons*
  - 5. Write: restore the value
- Refresh
  - 1. Just do a dummy read & restore to every cell.



Classical DRAM Organization

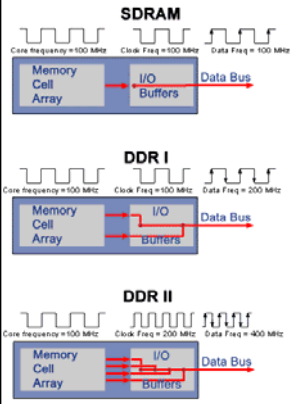


- Row and Column Address together:
  - Select 1 bit a time

DRAM Optimizations

- Fast Page Mode: consecutive accesses to same row
  - Row once, vary column address
  - Row access slow but varying columns fast
- EDO DRAM: Extended data out
  - FPM plus pipelining of column accesses
- Synchronous DRAM
  - Tied to system clock, increasing bus-speed
  - SDRAM-DDR, DDR-2, DDR-3
- RDRAM (Rambus)

DRAM organizations



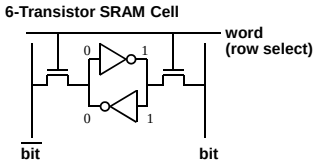
- DRAM core unchanged
- Organization/data transfer optimizations
- Compare SDRAM vs. DDR vs DDR2

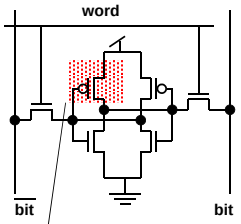
- *Picture source: http://www.lostcircuits.com/ via xbitlabs.com*

SRAM

- Data is static (as long as power is applied) (static != non-volatile)
- Logically, two cross-connected inverters with switches
  - Two inverters in a loop which remembers until next write and as long as power is on
  - CMOS inverter, MOS switch
  - 6-transistor implementation

6T SRAM Cell

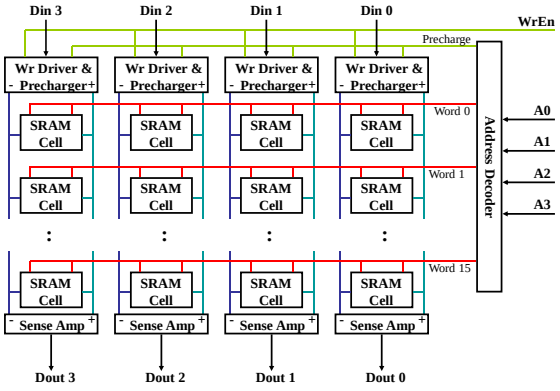




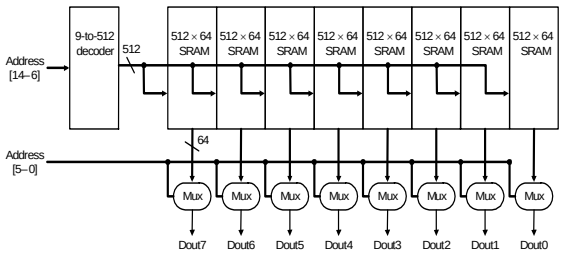
- Write:
  1. Drive bit lines (bit=1, $\overline{bit}$=0)
  2.. Select row
- Read:
  1. Precharge bit and $\overline{bit}$ to Vdd
  2.. Select row
  3. Cell pulls one line low
  4. Sense amp on column detects difference between bit and $\overline{bit}$

5T version: replaced with pullup to save area

SRAM Organization (16x4)



SRAM Organization



- Internal arrays may be different
  - 32Kx8 array realized with 8 512x64 arrays

Memory Technology Summary

- Performance of Main Memory:
  - Latency: Total delay seen
    - *Access Time*: time between request and word arrives
    - *Cycle Time*: time between requests
  - Bandwidth: I/O & Large Block Miss Penalty (L2)
- Main Memory is *DRAM*: Dynamic Random Access Memory
  - Dynamic since needs to be refreshed periodically (8 ms)
  - Addresses divided into 2 halves (Memory as a 2D matrix):
    - *RAS* or *Row Access Strobe*
    - *CAS* or *Column Access Strobe*
- Cache uses *SRAM*: Static Random Access Memory
  - No refresh (6 transistors/bit vs. 1 transistor /bit)
  - Address not divided
- *Size*: DRAM/SRAM *256-512,*
- *Cost/Cycle time*: SRAM/DRAM *32-64*

Technology Trends

DRAM		
Year	Size	Cycle Time
1983	256 Kb	220 ns
1986	1 Mb	190 ns
1989	4 Mb	165 ns
1992	16 Mb	145 ns
1995	64 Mb	120 ns
1998	256 Mb	100ns
2001	1 Gb	60ns
2004	4 Gb	50ns**
2008	16 Gb	45ns
2014	64Gb	45ns

Capacity Speed (latency)

Logic: 2x in 3 years 2x in 3 years-Not any more!

DRAM: 4x in 3 years 2x in 10 years

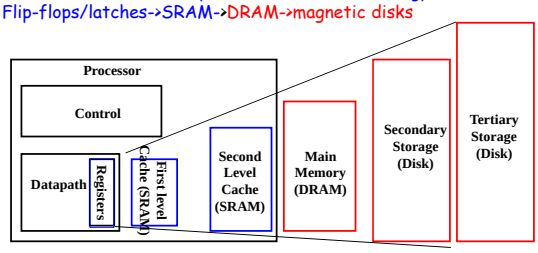
Disk: 4x in 3 years 2x in 10 years

Challenge: CPU-Memory Gap

- Fact: Large memories are slow (and cheap), fast memories are small (and expensive)
- How do we create a memory that is large, cheap and fast (most of the time)?
  - Hierarchy
  - Parallelism

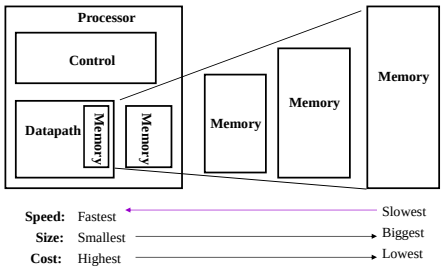
The Memory Hierarchy

- By taking advantage of the principle of locality:
  - Present the user with capacity of the cheapest technology.
  - Provide access at the speed of the fastest technology.
- Flip-flops/latches->SRAM->DRAM->magnetic disks

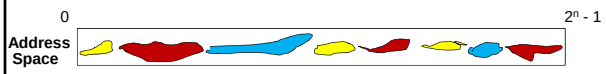


Speed (ns):	1s	5-10s	50-100s	10,000,000s (10s ms)	10,000,000,000s (10s sec)
Size (bytes):	100s	Ks(~Ms)	~Gs	Gs/Ts	Ts

Tradeoffs

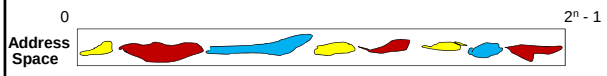


Locality



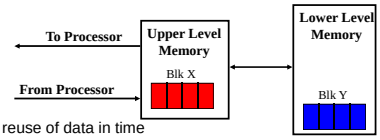
- The Principle of Locality:
  - Programs access small portions of address space for some period of execution time before moving to other portions
    - Blue portions for a while, THEN red portions for a while, THEN yellow etc
    - KEY: Portions need NOT be (often is not) contiguous but total size MUCH SMALLER than entire address space
- A library
  - Finding the few books you want: Slow
    - Reading various chapters: Fast
    - Switching between books: Fast
  - Library -> Memory: Larger the better
  - Books at table -> Cache: Size is limited but access is faster
- World will stop without locality!!

Locality



- WHY does locality exist?
- Think about code – what in code makes this happen?

Two flavors of locality



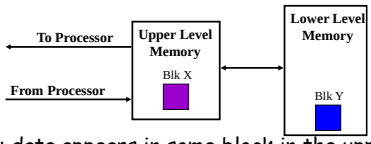
Temporal locality – reuse of data in time
Spatial locality – use of nearby data

- Temporal Locality (Locality in Time):
  - ⇒ Keep most recently accessed data items closer to the processor
  - ⇒ Odds are you'll refer to books on your table more than once
- Spatial Locality (Locality in Space):
  - ⇒ Move blocks consists of contiguous words to the upper levels
  - ⇒ Odds are you'll read contiguous pages/chapters

Connect Locality & caches

- Locality says data much smaller than total memory accessed in a time window
- Put that smaller data in a small, fast memory called cache
  - Move data to cache upon access (temporal)
  - Move a block instead of one word (spatial)
- As locality changes over time, move new data from main memory & replace old data in cache

Illusion of Speed and Capacity



- Hit: data appears in some block in the upper level (example: Block X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: Time to access the upper level which consists of RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieved from a block in the lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor
- Hit Time << Miss Penalty

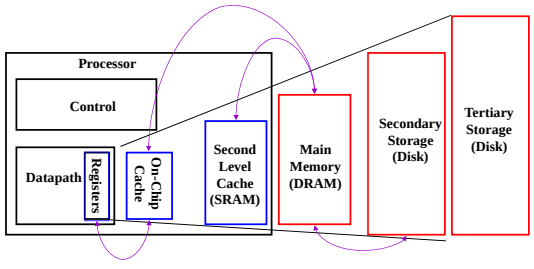
Why caches work

- Ahmdahl's law and caches
  - Big win if: cache-hits are common case
  - Property of programs: "Locality"
- Average memory access time
  - F(hit-time, miss-rate, miss-penalty)
  - Hit time + miss rate * miss penalty
  - 2 ns + 4% * 50 ns = 2 + 2 = 4 ns
    - versus 50 ns if no caching

Lookahead: Caches

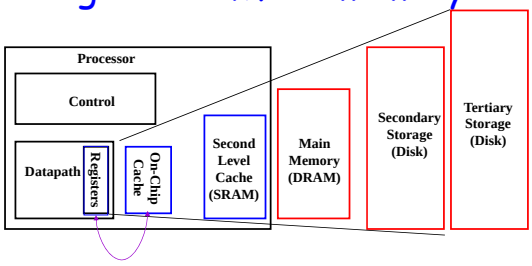
- Several issues:
  - Determine hit/miss: How do we know if a data item is in the cache?
  - If it is, how do we find it?
  - If it isn't, where do I place it?
  - Replacement: What do we do with data that was present?
  - Who manages this? Compiler? Hardware? Software/OS/Programmer?

Managing the memory hierarchy



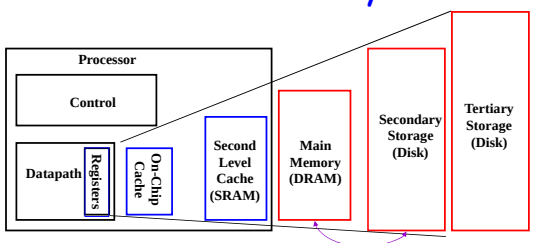
- Whose responsibility is it?
  - Short answer: it depends on the level

Register<->Main memory



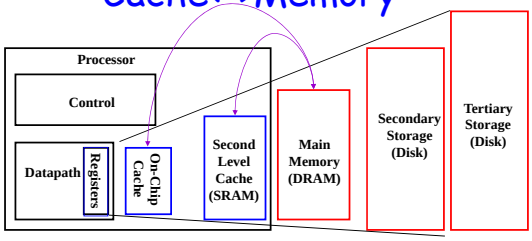
- Managed by compiler (sometimes programmer)
  - "Word" granularity
  - Load/store ties memory locations to registers (allocation)
  - Register temporaries ("spill" to memory when needed)
- Complexity!

Disk<->Memory



- Managed by OS for memory: Automatic & transparent to user
  - Virtual memory
  - Illusion of large memory, protection
  - More later
- Managed by programmer for files: Explicit file read/write

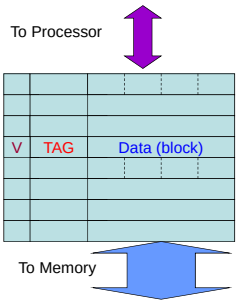
Cache<->Memory



- Hardware managed: needs to be fast
- Automatic: to avoid complexity of explicit management
- "Block" granularity to exploit spatial locality
- Retain recently accessed blocks to exploit temporal locality

Cache Operation

- Tag, data, valid
- Tag:
  - Mapping many to few (full address space to smaller cache)
  - Many to few → many addresses map to same slot in cache → tag says which specific address is in a given slot
- Data:
  - Block: more than one word
- Valid:
  - Not everything in cache is meaningful
- Frame (block-frame/cache-frame)

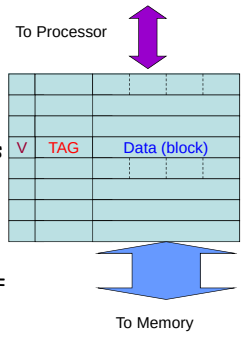


Cache Operation

- Hit/Miss detection
  - If (incoming tag == stored tag)
    - Hit //i.e. block is resident in cache
    - Return word to processor
  - Else
    - Miss
      - Make space : replace some other block
      - Get block from memory
      - Put block in "data" part, set tag using new address tag

Example of cache operation

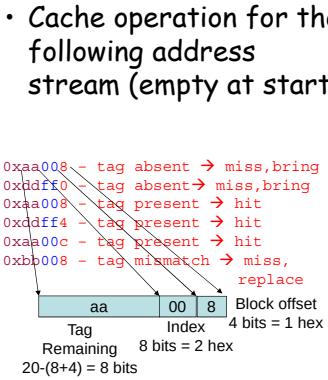
- 256-frame cache
- 20 bit address-space
- 16 byte blocks
- Use***:
  - 16-byte block: Low 4 addr bits within block ("block offset")
  - 256 frames → next 8 addr bits as "index" of frame (many-to-few via simple modulo mapping – modulo 256 = next 8 address bits)
  - Why such simple mapping?
  - All other addr bits as "tag"



Cache Operation

- Cache operation for the following address stream (empty at start)

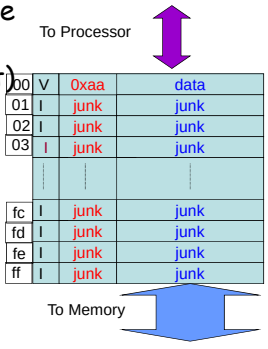
```
0xaa008 – tag absent → miss,bring
0xddff0 – tag absent→ miss,bring
0xaa008 – tag present → hit
0xddff4 – tag present → hit
0xaa00c – tag present → hit
0xbb008 – tag mismatch → miss,
                         replace
```

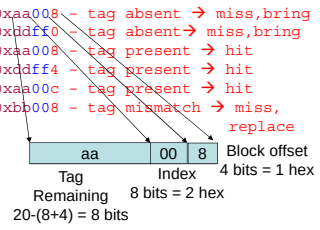


Cache Operation

- Cache operation for the following address stream (empty at start)

```
0xaa008 – tag absent → miss,bring
0xddff0 – tag absent→ miss,bring
0xaa008 – tag present → hit
0xddff4 – tag present → hit
0xaa00c – tag present → hit
0xbb008 – tag mismatch → miss,
                         replace
```

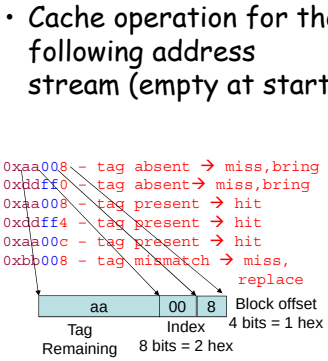




Cache Operation

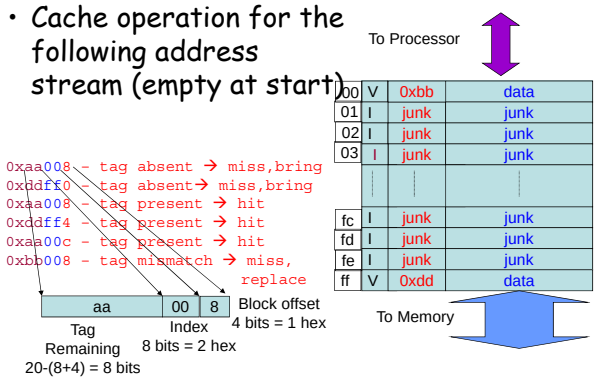
- Cache operation for the following address stream (empty at start)

```
0xaa008 – tag absent → miss,bring
0xddff0 – tag absent→ miss,bring
0xaa008 – tag present → hit
0xddff4 – tag present → hit
0xaa00c – tag present → hit
0xbb008 – tag mismatch → miss,
                         replace
```



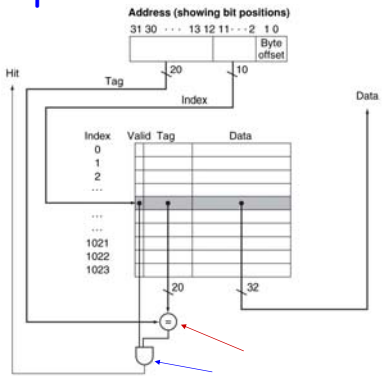
Cache Operation

- Cache operation for the following address stream (empty at start)

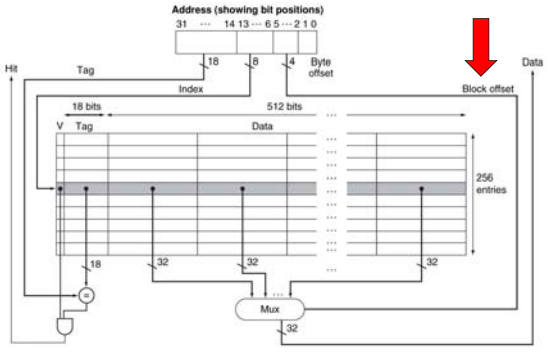


4 KB, 4-byte block Cache Operation

- Tag comparators
- Hit detection
- How many frames?
- What is the cache size?
- What if block size is more than one word?



16KB size, 64-byte block



- Use block-offset to select word (4 bytes) out of a block

Byte offset vs. Block offset

- Careful – both abbreviated as BO!
- Block offset is word/half-word/byte within block
- Byte offset is byte within word
- Block offset includes byte offset because addresses are byte addresses
  - See previous slide

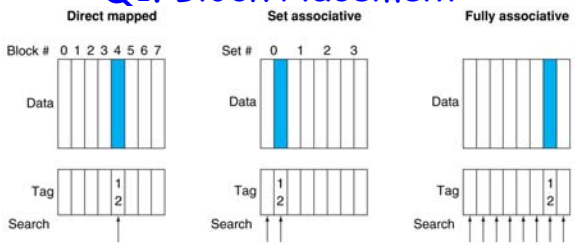
Checkpoint

- Summary:
  - Cache management in hardware
  - Caches terminology and organization
    - Frames
    - Blocks
    - Tags
  - Example of Cache operation
- Next: 4 questions
  - Where is a block placed?
  - How is a block found?
  - Which block is replaced?
  - What happens on a write?

4 Questions for Memory Hierarchy

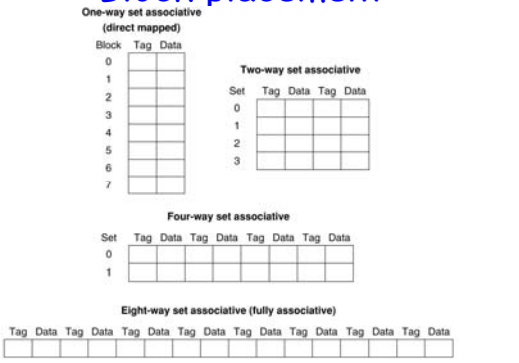
- Q1: Where can a block be placed in the cache? *(Block placement)*
- Q2: How is a block found if it is in the cache? *(Block identification)*
- Q3: Which block should be replaced on a miss? *(Block replacement)*
- Q4: What happens on a write? *(Write strategy)*

Q1. Block Placement



- 12 above is DECIMAL (book eg)!!!!
- In previous example – direct mapped: MANY TO ONE
  - Block may reside in ONLY ONE frame. (frame# = modulo #blocks)
- Fully-associative: MANT TO ANY
  - Block may reside in ANY frame (no modulo needed)
- N-way set-associative: MANY TO N
  - Block may reside in a SET OF N frames (set # = addr modulo #sets)
  - Search all N frames within a set

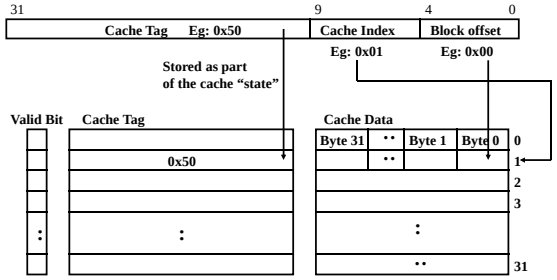
Block placement



- 8 blocks – directmapped, 2-way , fully-assoc (ONE set in ENTIRE cache)

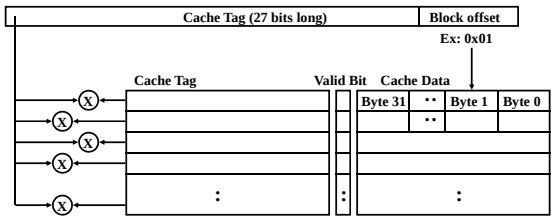
Block Identification Example: 1 KB Direct Mapped Cache with 32 B Blocks

- Break address into block offset, index, tag
- 32 byte blocks → 5 bits block offset [0-4]
- 1KB and 32 byte blocks → 32 blocks/frames → 5 bits index [5-9]
- Rest tag [10-31]



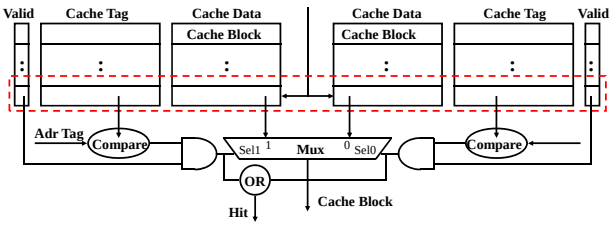
Another Extreme Example: Fully Associative

- Fully Associative Cache
  - ONE set in entire cache – so how many index bits?
  - Compare the Cache Tags of all cache frames in parallel
  - Example: Block Size = 32 B blocks → 5 bits of block offset and rest tags

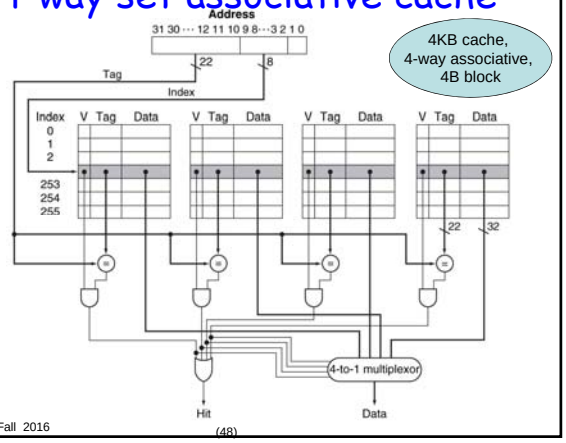


A Two-way Set Associative Cache

- N-way set associative: N blocks in each set
  - N tags in set searched in parallel
- Example: Two-way set associative cache
  - Cache Index selects a "set" from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result



4-way set associative cache



Terminology

- Cache block – aka cacheline (I use "block")
- way – one of the associative ways in a set
- Set- set of all the ways/blocks in a set
  - A bit circular but you get the point!
  - E.g., 4 ways/blocks in a 4-way assoc set
- Index – selects the set
- Frame – physical location for 1 cache block

Organization Methodology

- How to determine:
  - Number of bits for
    - Index, tag and block offset
- Walkthrough example(s)
  - 32KB, 32B block, 2-way associative cache

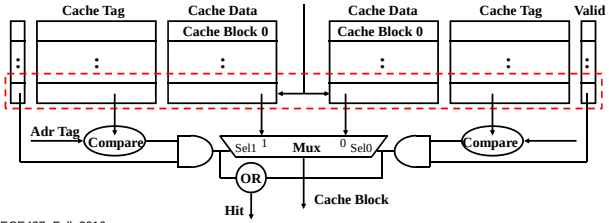
Cache Organization

- Cache size = 32 KB (CS)
- Block size = 32 B (BS)
  - Frames (F) = CS/BS = 1024 (= 1K)
- Associativity = 2-way (A)
  - #sets N = Number of frames/way = F/A = 512

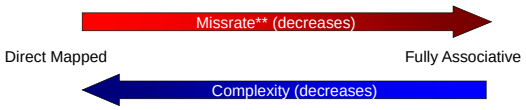
- Address-bits = 32 bits (Ad)
  - Block-offset bits (b) = lg(BS) = lg(32) = 5
  - Index bits (i) = lg(N) = lg(512) = 9
  - Tag bits (t) = Ad – i – b = 32 – 9 – 5 = 18

Disadvantage of Set Associative Cache

- N-way Set Associative Cache versus Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data – LATE mux select signal
  - Data comes AFTER Hit/Miss decision and set selection
- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue. Recover later if miss.

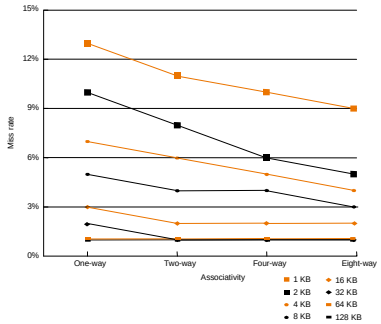


Associativity spectrum



- "Conflict misses" reduced with higher associativity
  - Misses due to cache mapping
- Associative search is complex
- Increasing associativity:
  - Increases tag bits, shrinks index bits
  - Increases comparator size (~ tag bits)

Performance



- A little associativity goes a long way

Checkpoint and Lookahead

- Q1 and Q2: Block placement & id
  - Simple case : direct mapped
  - Associativity: trade-offs
  - Cache implementation
- Next:
  - Quick recap
  - 3C's : Miss classification
  - Q3: Replacement
  - Q4: Write strategies
  - How to design memory hierarchies?
  - How does software interact with caches?
  - Is programmer aware of the existence of caches?
  - Can programmers benefit by being aware of caches?

Recap Q1 & Q2

- Q1: Where can a block be placed in the upper level? *(Block placement)*
  - In one of N-frames in N-way associative cache
  - N = 1 => Direct mapped
  - N = #frames => Fully associative
  - Setindex = Blocknum (mod numsets)
- Q2: How is a block found if it is in the upper level? *(Block identification)*
  - Tag match (no need to examine index/block-offset bits --- why?)
  - Valid bit

Recap: Cache Block Diagrams

- 96KB, 3-way set associative, 64Byte block cache
- Direct-mapped, 16KB, 128 byte block cache

Recap: Exercise

- Draw the block diagram of a word-addressable 64 KB, 2 way set-associative cache with 128 byte blocks

Miss Classification

- Compulsory (cold start or process migration, first reference): first access to a block
  - "Cold" fact of life: not a whole lot you can do about it (true?)
  - Note: If you are going to run "billions" of instruction, Compulsory Misses are insignificant
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- Capacity:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size
- Coherence (invalidations): other process (e.g., in multicores, I/O) updates memory – ch. 7

Source of Cache Misses Quiz

	Direct Mapped	N-way Set Associative	Fully Associative
Cache Size: Small, Medium, Big?	Big	Medium	Small
Compulsory Miss:			
Conflict Miss			
Capacity Miss			
Coherence Miss			

Choices: Zero, Low, Medium, High, Same
Size inferred in Capacity miss row

Sources of Cache Misses Answer

	Direct Mapped	N-way Set Associative	Fully Associative
Cache Size	Big	Medium	Small
Compulsory Miss	Same	Same	Same
Conflict Miss	High	Medium	Zero
Capacity Miss	Low	Medium	High
Coherence Miss	Same	Same	Same

Note:
If you are going to run "billions" of instruction, Compulsory Misses are insignificant.
Size inferred in Capacity miss row.

Q3. Block Replacement

- Q3: Which block should be replaced on a miss?
  (Block replacement)
  - Easy for Direct Mapped
  - Set Associative or Fully Associative:
    - Random
    - FIFO (Not a good idea, in general)
    - LRU (Least Recently Used)
    - NRU (Not Recently Used)

Smart

Not Stupid

OPT (Ideal)
LRU
NRU

Simple heuristics capture large fraction of opportunity.
Sweat for the last 10%

Exercise

- Give an example of an address stream where
  - 2-way associative cache is better than direct-mapped cache
  - Direct mapped cache is better than 2-way cache.
- Use 16 entry caches, assume LRU replacement

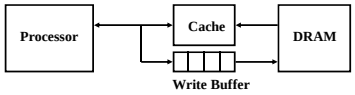
Q4: Writes

- Read read tag and data in parallel BEFORE tag match
- Writes cannot do that
  - If you write to a non-matching block, data will be lost
  - So writes have to read and check tag FIRST and then write to the matching block
  - Typically done in a pipelined manner – previous write writes to data while next write reads and checks tag

Q4. Write strategy

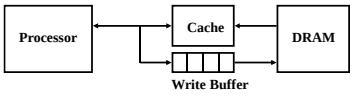
- Q4: What happens on a write? *(Write strategy)*
- *Write through*—The information is written to both the block in the cache and to the block in the lower-level memory.
- *Write back*—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each?
  - WT: read misses cannot result in writes
  - WB: no repeated writes
- WT always combined with write buffers so that don't wait for lower level memory

Write Buffer for Write Through

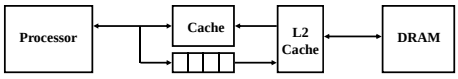


- A Write Buffer is needed between the Cache and Memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4-8
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
  - Can tolerate bursty behavior
- Memory system designer's nightmare:
  - Store frequency (w.r.t. time) close to 1 / DRAM write cycle
  - Write buffer saturation

Write Buffer Saturation

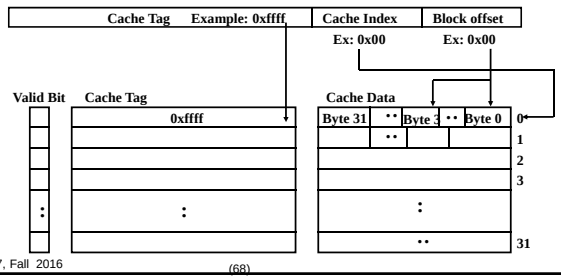


- Store frequency (w.r.t. time) > 1 / DRAM write cycle
  - If this condition exists for a long period of time (CPU cycle time too quick and/or too many store instructions in a row):
    - Store buffer will overflow no matter how big you make it
    - CPU Store Cycle Time <= DRAM Write Cycle Time
- Solution for write buffer saturation:
  - Use a write back cache
  - Install a second level (L2) cache:

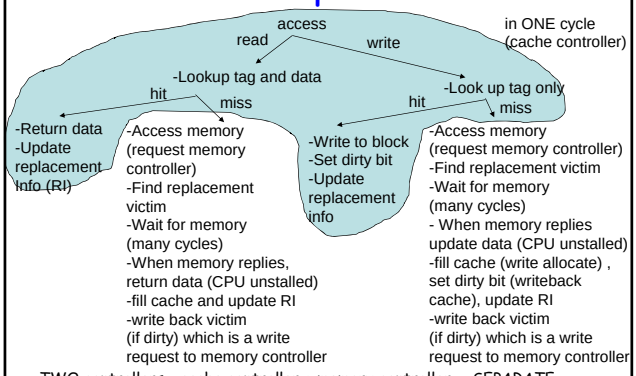


Write-miss Policy: Write Allocate versus Not Allocate

- Assume: a 16-bit write causes a miss
  - Do we put the block in the cache?
    - Yes: Write Allocate (read block from memory, overwrite word/byte – no holes in blocks in cache)
    - No: Write No Allocate (write word/byte AROUND cache)



Cache operation



- TWO controllers – cache controller + memory controller - SEPARATE
- ALL actions by cache controller (memory controller acts on cache's requests)

Cache operation

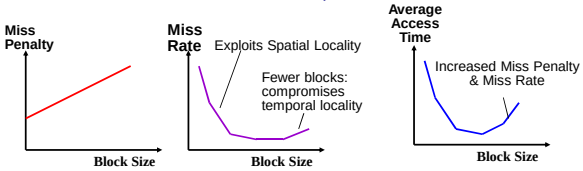
- Book has a 4-state state machine for cache operation
  - CAREFUL – Idle state unnecessary and each state takes a cycle so it is slower than it needs to be (hits may not be one cycle – Idle state THEN Hit state)
- Previous slide – hits are one cycle WITHIN cache controller (shaded)
  - fits within IF and MEM stages of pipeline
  - Misses are multiple cycles
    - Cache controller requests memory controller and waits for reply
    - Miss path is multiple states (cycles) but hits are one state (one cycle)
    - hits are combinational based on tag and miss path is multiple states

Cache operation

- IF/MEM stages request cache controller
  - Separate instruction- and data-caches (later)
  - Without cache, MEM directly requests memory controller
  - Cache controllers ask memory controller on a miss – SEPARATE cache and memory controllers (if you merge these you will suffer later)
- Previous slide is one example – you can modify to suit your design

Block Size Tradeoff

- In general, larger block size take advantage of spatial locality BUT:
  - Larger block size means larger miss penalty:
    - Takes longer time to fill up the block
  - If block size is too big relative to cache size, miss rate will go up
    - Too few cache blocks
- In general, Average Access Time:
  - = Hit Time + Miss Penalty x Miss Rate



Block Size

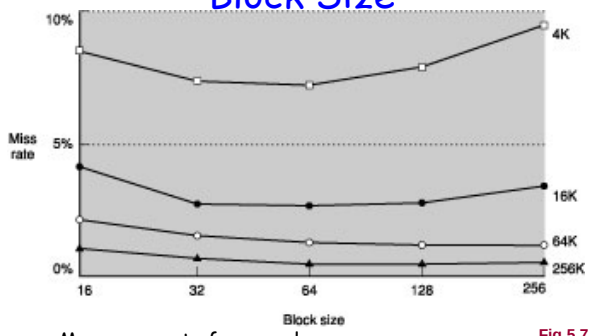
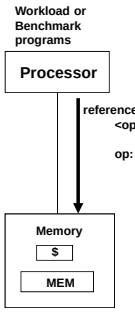


Fig 5.7

- Measurements from real programs
- Bottomline: Block size chosen by experiment, typically 16-128 bytes

Big picture



Optimize the memory system organization to minimize the average memory access time (AMAT) for typical workloads

- Why do we care about AMAT? AMAT affects CPI
  - Remember there is 1.x memory ops per instruction

Impact on Performance

- Suppose a processor executes at
  - Clock Rate = 2 GHz (0.5 ns per cycle)
  - Ideal CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control
- Suppose that 5% of memory operations get 100 cycle (50ns) miss penalty
- CPI = ideal CPI + average stalls per instruction
  = 1.1(cyc) +(0.30 (datamops/ins)
  x 0.05 (miss/datamop) x 100 (cycle/miss))
  = 1.1 cycle + 1.5 cycle
  = 2. 6
- ~58 % of the time the processor is stalled waiting for memory!
- A 0.5% instruction miss rate would add an additional 0.5 cycles to the CPI

AMAT Impact quiz

	Cache hit time	Miss penalty	Missrate
Cache Size: Small, Medium, Big?			
Associativity: Low-to-high			
Block size: Low to high			
Replacement Policy FIFO to LRU			

- Average Memory Access time
- AMAT = hit time + miss-rate * miss-penalty
- Choices: same, increasing or decreasing
- Ignore last row for now

AMAT Impact: Answers

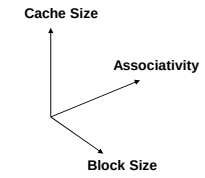
	Cache hit time	Miss penalty	Missrate
Cache Size: Small, Medium, Big?	inc	same	dec
Associativity: Low-to-high	inc	same	dec
Block size: Low to high	same	inc	dec**
Replacement Policy FIFO to LRU	same	same	dec*

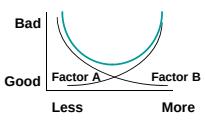
Improving Cache Performance

- Reduce Hit time
  - small and simple-> direct mapped
- Reduce miss rate
  - Large cache, large block size, associative,
- Reduce miss penalty
  - Reduce block-size
- Remember Amdahl's law
  - Common case : hit
  - Reduce miss-rate at the cost of hit time

Cache design space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins

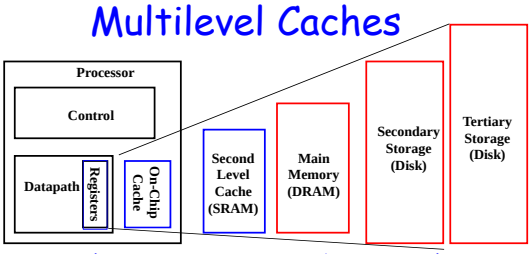




Practical design issues

- Multi-level Caches
- Split Cache vs. unified cache

Multilevel Caches



- $AMAT_{L1}$ = $hit\ time_{L1}$ + $miss\text{-}rate_{L1}$ * $miss\text{-}penalty_{L1}$
- What is $miss\text{-}penalty_{L1}$?
  - Access time of memory: Huge and RELATIVELY increases as CPU clock/L1 speed doubles while memory speed improves 5%
- Put in a large L2 cache between L1 and memory to fill the huge gap between CPU/L1 and memory
  - What is the $miss\text{-}penalty_{L1}$?
  - $AMAT_{L2}$ = $hit\ time_{L2}$ + $miss\text{-}rate_{L2}$ * $miss\text{-}penalty_{L2}$

Multilevel Caches

- Cycle time = 0.5ns (~ 2GHz clock)
- Main memory access = 50ns = 100 cycles
- L1 miss rate = 5%
- Without 2nd level cache
  - $AMAT_{L1}$ = 1 + 5% * 100 = 6 cycles
- With 2nd level cache
  - L2 miss-rate = 2% (local miss-rate)
  - L2 hit time = 10 cycles
  - $AMAT_{L2}$ = 10 + 2% * 100 = 12 cycles
  - $AMAT_{L1}$ = 1 + 5% * 12 = 1.6

AMAT Impact: Answers

	Cache hit time	Miss penalty	Missrate
Cache Size: Small, Medium, Big?	inc	same	dec
Associativity: Low-to-high	inc	same	dec
Block size: Low to high	same	inc	dec**
Replacement Policy FIFO to LRU	same	same	dec*
Multi-level Caches	same	dec*	same

Split caches

- One for instruction, one for data
- Split cache
  - Instructions account for 75% of mem accesses
  - I-missrate = 5%, D-missrate = 6%
  - AMAT = (1 + 0.05*10)*0.75 + (1 + 0.06*10) * 0.25
  - = 1.525
- Unified Cache
  - Aggregate missrate = 4%
  - AMAT = (1 + 0.04*10) = 1.4???
  - For modern pipelined processor:
    - single-memory structural hazard

State of the art

- 2-3 levels of cache (SRAM)
- Split I- and D-caches at Level 1
- Low associativity at Level 1
- Higher associativity at subsequent levels

Real stuff – A8 and i7

Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

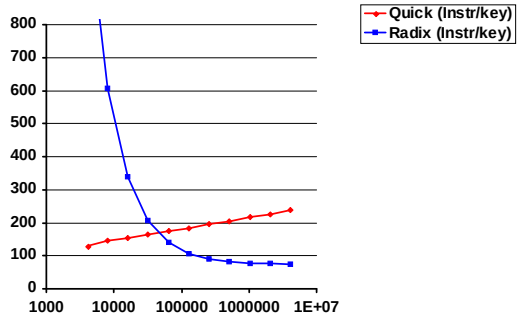
Summary

- Memory technology (Capacity/cost/speed)
- Need for hierarchy
- Performance
  - AMAT, ideal vs. real CPI
- Cache management:
  - Associativity, indexing, write handling, multi-word blocks etc.
- Diagrams of arbitrary cache organizations
- Next:
  - Cache-friendly programming techniques
  - Virtual Memory

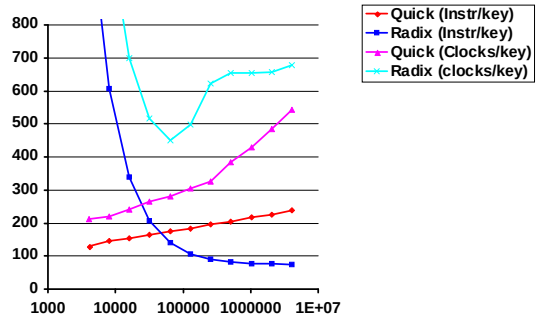
Software interaction

- RAM model of computation
  - All memory accesses - take the same amount of time as an add = 1 unit
  - Theoretical Model "random access memory" – has nothing to do with DRAM or SRAM
- Reality:
  - Caches introduce non-uniformity
  - Hits faster than misses
- Quicksort $\Theta(n \lg(n))$
  - fastest comparison based sorting algorithm when all keys fit in memory:
- Radixsort $\Theta(n)$
  - also called "linear time" sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys:

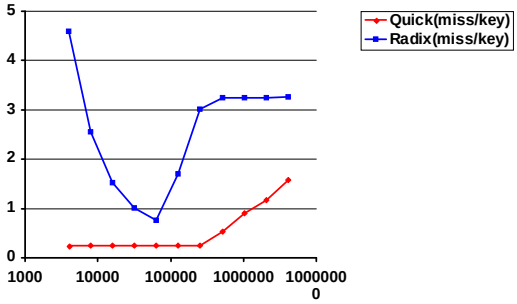
QS vs. RS : Instructions



QS vs. RS : Time, Instructions



QS vs. RS : Cache misses



- RAM model results are still valid… but at much larger input sizes
- How does one create practical, fast algorithms?
- Cache-aware programming/compilation

Exercise 1

- What is the miss-rate of the following code segment for
  - 16 frame, direct mapped cache with 4-word cache blocks?
  - 16-frame, 2-way set-associative cache with 4-word cache blocks?
- Assume cold start (i.e. no valid data in the cache to begin with)
- Assume the following start addresses (bytes) for the three arrays
  - A : 0xC000
  - B : 0xC100
  - C : 0xC200
- Ignore writes for now

```
for(i=0;i<64;i++) {
    C[i] = min( A[i] , B[i] );
}
for(i=0;i<64;i++) {
    C[i] = max( A[i] , B[i] );
}
```

What if start addresses are:
A : C000
B : D110
C : E220
instead?

Worksheet 1

Cache 1

T=8	I=4	BO=4
C0	0	0
C1	0	0

Cache 2

T=9	I=3	BO=4
C0	0	0
C1	0	0

Address Stream	Cache 1	Cache 2
C000 (A[0])	Miss (compul)	Miss
C100 (B[0])	Miss (compul)	Miss
C004 (A[1])	Miss (conflict)	Hit
C104 (B[1])	Miss (conflict)	Hit
C008 A[2]	Miss (conflict)	Hit
C108 B[2]	Miss (conflict)	Hit
C00C A[3]	Miss (conflict)	Hit
C10C B[3]	Miss (conflict)	Hit
C010 A[4]	Miss (compul)	Miss
C110 B[4]	Miss (compul)	Miss

[0-3] – 1 block
[4-7] – 1 block
[8-11] – 1 block etc

A and B conflict on every block in direct-mapped

A and B blocks sit in different ways in 2-way assoc → no conflict

Cache-aware programming

- Instruction Sequencing
  - *Loop Interchange*: change nesting of loops to access data in order stored in memory
  - *Loop Fusion*: Combine 2 independent loops that have same looping and some variables overlap
  - *Tiling (blocking)*: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down entire columns or rows
- Data Layout
  - *Merging Arrays*: Improve spatial locality by single array of compound elements vs. 2 separate arrays
  - *Nonlinear Array Layout*: Mapping 2 dimensional arrays to the linear address space
  - *Pointer-based Data Structures*: node-allocation
- Example walkthrough: Loop fusion, Tiling (blocking), Merging Arrays

Worksheet for example

	Base	Loop Fusion	Array Merging	Tiling
Direct Mapped				
2-way Set associative				

- Count read misses

Worksheet for example

	Base	Loop Fusion	Array Merge	Tiling
Direct Mapped	64 (A, 1st loop) +64 (B,1st loop) +64(A,2nd loop) +64(B,2nd loop)	64+64+ 64+64	32+0+ 32+0	32+32+32 +32+ 32+32+32 +32
2-way assoc	16+16+16+16	16+16 +0+0	32+0+ 32+0	8+8+0+0+ 8+8+0+0

- Count read misses

Miss categorization

- Total A+B = 64+64 = 128 words
- Each miss brings in 4 words
  - Minimum of 32 (=128/4) misses
  - Cannot do better (without prefetch)
- Exercise: Identify conflict and capacity misses

Walk through Exercise 1

- Count read misses for the following code segment for
  - 16 entry, direct mapped cache with 4-word cache blocks?
  - 16-entry, 2-way set-associative cache with 4-word cache blocks?
- Assume cold start (i.e. no valid data in the cache to begin with)
- Assume the following start addresses for the three arrays
  - A : 0xC000
  - B : 0xC100
  - C : 0xC200
  - D : 0xC300
- Ignore writes for now
- USE WORKSHEET

```
for(i=0;i<64;i++) {
    C[i] = min( A[i] , B[i] );
}
for(i=0;i<64;i++) {
    D[i] = max( A[i] , B[i] );
}
```


#1 : Loop Fusion

- Converts distant reuse to near reuse
- Enhances temporal locality – reduces capacity misses
- Code Transformation

```
for(i=0;i<64;i++) {
    C[i] = min( A[i] , B[i] );
}
for(i=0;i<64;i++) {
    D[i] = max( A[i], B[i] );
}
```

```
for(i=0;i<64;i++) {
    C[i] = min( A[i] , B[i] );
    D[i] = max( A[i], B[i] );
}
```


#2: Array merging

- Eliminates conflicts
  - Array of compound structure vs.
  - multiple arrays of simple data
- Enhances spatial and temporal locality
- Data layout transformation

```
for(i=0;i<64;i++) {
    C[i] = min( A[i] , B[i] );
}
for(i=0;i<64;i++) {
    D[i] = max( A[i], B[i] );
}
```

```
Struct merge {
    int  A;
    int  B;
};
Struct merge M[64];
for(i=0;i<64;i++) {
    C[i] = min( M[i].A , M[i].B);
}
for(i=0;i<64;i++) {
    D[i] = max( M[i].A, M[i].B);
}
```

#3: Blocking (Tiling)

- Exploits re-use across loops
  - Divide into pieces that fit in the cache vs.
  - Marching through whole array
- Capacity misses
- Code Transformation

```
for(i=0;i<64;i++) {
    C[i] = min( A[i] , B[i] );
}
for(i=0;i<64;i++) {
    D[i] = max( A[i], B[i] );
}
```

```
for (j=0; j<2;j++)
{
 for(i=0;i<32;i++) {
    C[32*j + i] = min( A[32*j + i] , B[32*j + i] );
 }
 for(i=0;i<32;i++) {
    D[32*j + i] = max( A[32*j+ i], B[32*j + i] );
 }
}
```


State of the practice

- Cache friendly programming challenges
  - No global view of application
  - Different cache sizes
- Analyze programs after they're written
  - Find bad access patterns
  - Fix them
  - Lather, Rinse and Repeat

Recap: Data Cache Performance

- Instruction Sequencing
  - *Loop Interchange*: change nesting of loops to access data in order stored in memory
  - *Loop Fusion*: Combine 2 independent loops that have same looping and some variables overlap
  - *Blocking*: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down entire columns or rows
- Data Layout
  - *Merging Arrays*: Improve spatial locality by single array of compound elements vs. 2 separate arrays
  - *Nonlinear Array Layout*: Mapping 2 dimensional arrays to the linear address space
  - *Pointer-based Data Structures*: node-allocation
- Example walkthrough: Loop fusion, Blocking, Merging Arrays
- Ch 5a done