Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

Cache Memory

Schedule

- **Today**
  - Chapter 6 – Memory systems (caches)
- **Friday and Monday**
  - Chapter 6 – Virtual memory
- **Next Wednesday**
  - Chapter 7 – Input / Output Systems
- **Next Friday (March 30th)**
  - **Quiz #5** – Chapter 6 (Cache & Virtual Memory)

Recap – Searching Cache for Address "A"

Direct-Map Cache

- Decode A into tag, block, and offset bits
- Go to the block
  - Does tag in block match search tag?
  - Is valid bit set?
- If yes to both, **match**!
  - Retrieve data from offset in block

Fully Associative Cache

- Decode A into tag and offset bits
- Search **all** blocks in cache for matching tag (where valid bit is also set
- If match
  - Retrieve data from offset in matching block

Recap – Searching Cache for Address "A"

Set-Associative Cache

- Decode A into tag, set, and offset bits
- Go to the set (of several blocks)
  - Search **all** blocks in the **set**
  - Does tag in block match search tag?
  - Is valid bit set?
- If yes to both, match!
  - Retrieve data from offset in block

Cache Replacement Policies

Replacement Policy

- In a fully associative or set associative cache, a **replacement policy** ("algorithm") is run whenever we need to evict a block from cache
- **What would the perfect replacement policy be?**
  - Look into the future to see which blocks won't be needed for the longest period of time – evict those first!
  - This is often called the "*oracle*", as in a prophet…
- The perfect replacement policy is **impossible to implement (unless you have a time machine)**, but it serves as a benchmark to compare actual implementable algorithms against

Replacement Policy

- Algorithm 1
  - **Least recently used (LRU)**
  - Keeps track of the last time that a block was assessed in the cache
  - Evict the block that has been unused for the longest period of time
- **Drawbacks?**
  - Complexity! RU has to maintain an access history for each block, which ultimately slows down the cache

Replacement Policy

- Algorithm 2
  - **First-in, first-out (FIFO)**
  - The block that has been in the cache the longest is evicted, regardless of when it was last used
- **Strengths and weaknesses?**
  - Strengths – Easier to implement
  - Weaknesses – The oldest block in the cache might be the most popular!
    - *If we evict it and it is popular, we'll get it back in the cache soon enough…*

Replacement Policy

- Algorithm 3
  - **Random replacement**
  - Picks a block at random and replaces it with a new block
- **Strengths and weaknesses?**
  - Strengths – Simple to implement. Never thrashes
  - Weaknesses – Might evict a block that will be needed often or needed soon

Cache and Writing

- Up to now, we have talked about reading from main memory
  - And getting faster reads via the cache!
- What about writing to main memory?
  - **Can we get faster writes with a cache?**
- Yes! We can write data not to main memory, but to the (faster) cache instead!

Cache and Writing

- Writing to the cache poses a problem, though
  - If the cache block has been modified from what is in memory, we can't just evict it when we need space – it must be written back to memory first
  - New term – "**Dirty" blocks**
    - Blocks that have been updated while they were in the cache but not written back to main memory yet
  - Cache replacement policies must take into account dirty blocks when deciding who (and how) to evict from the cache

Cache and Writing

Write Through

- Updates cache and main memory simultaneously on every write
- Pro – Simple!
- Con – slows down the access time on updates
  - Usually negligible because the majority of accesses tend to be reads, not writes

Write Back

- Updates memory only when the block is selected for replacement
- Pro – memory traffic is minimized
- Con – The value in memory does not always agree with the value in cache (causing problems in multi-core / multi-processor systems with many caches)

Memory Access Time

Effective Access Time

- The performance of hierarchical memory is measured by its effective access time (EAT)
- EAT is a weighted average
  - Takes into account the hit ratio and relative access times of successive levels of memory
- EAT for a two-level memory:
  - **EAT = H × AccessC + (1-H) × AccessMM**
    - H is the cache hit rate
    - AccessC and AccessMM are the access times for cache and main memory, respectively
  - This equation can be extended to any number of memory levels

Effective Access Time

- Example computer system
  - Main memory access time: 200ns
  - Cache access time: 10ns
  - Cache hit rate: 99%
- Suppose access to cache and main memory occurs **concurrently** (i.e. the accesses overlap)
- EAT = 0.99(10ns) + 0.01(200ns)
  = 9.9ns + 2ns
  = **11.9ns**

Effective Access Time

- Example computer system
  - Main memory access time: 200ns
  - Cache access time: 10ns
  - Cache hit rate: 99%
- Suppose access to cache and main memory occurs **sequentially** (i.e. the accesses do not overlap)
- EAT = 0.99(10ns) + 0.01(10ns + 200ns)
  = 9.9ns + 2.1ns
  = **12ns**

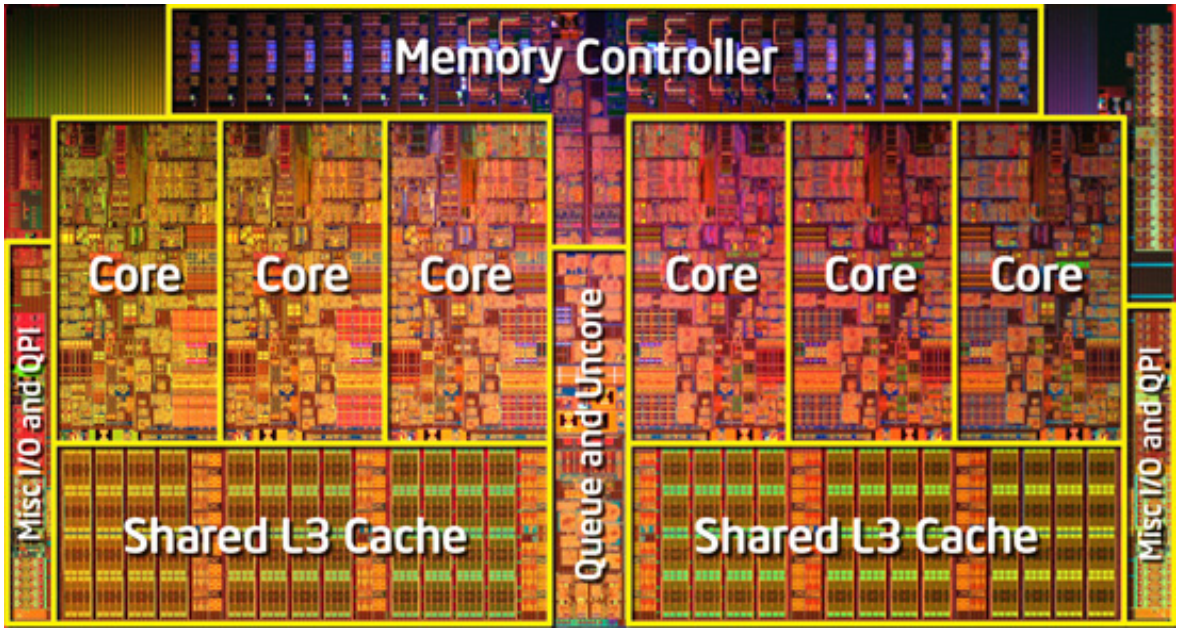
Cache Variations

Cache Variations

- Many variations on cache designs
- **Unified cache** – both instructions and data are cached together
- **Harvard cache** – separate caches for data and instructions
  - Provides better locality (i.e. performance) but increases complexity
  - Can get a similar benefit by simply providing a larger unified cache

Cache Example – Intel Core i7 980x

- High-end 6 core processor with a sophisticated multi-level cache hierarchy
- 3.5GHz, 1.17 billion transistors (!!!)



Cache Example – Intel Core i7 980x

- Each processor core has its own a L1 and L2 cache
  - 32kB Level 1 (L1) data cache
    - 8-way set associative, 64 byte block ("line") size
  - 32kB Level 1 (L1) instruction cache
    - 4-way set associative, 64-byte block size
  - 256kB Level 2 (L2) cache (both instruction and data)
    - 8-way set associative, 64-byte block size
- The entire chip (all 6 cores) **share** a single 12MB Level 3 (L3) cache
  - 16-way set associative, 64-byte block size

Cache Example – Intel Core i7 980x

- Access time? (Measured in 3.5GHz clock cycles)
  - 4 cycles to access L1 cache
  - 9-10 cycles to access L2 cache
  - 48 cycles to access L3 cache
- Smaller caches are faster to search
  - And can also fit closer to the processor core
- Larger caches are slower to search
  - Plus we have to place them further away

Cache Example – Intel Core i7 980x

- The Intel cache hierarchy is **inclusive**
  - All data in a smaller cache also exists at the next higher level
- Other vendors (e.g. AMD) have **exclusive** caches
  - Only 1 copy of the data in any cache (i.e. if it's in the L1 cache, it cannot also be in the L2 or L3 cache)
- Tradeoffs?
  - Circuit complexity
  - Wasted cache memory space