Efficiency: caching

COSC 208, Introduction to Computer Systems, 2021-11-01

Announcements

- Project 2 Part B due date extended to Tues, Nov 9

Outline

- Warm-up
- Instances of caching
- Cache replacement

Warm-up

Q1: *Cross-out unnecessary loads and stores from the assembly code.*

```
000000000000071c <multiply>:
    71c:    d10083ff    sub sp, sp, #0x20
    720:    b9000fe0    str w0, [sp, #12]       XXXXX
    724:    b9000be1    str w1, [sp, #8]        XXXXX
    728:    b9400fe1    ldr w0, [sp, #12]       XXXXX
    72c:    b9400be0    ldr w1, [sp, #8]        XXXXX
    730:    1b007c20    mul w0, w1, w0
    734:    b9001fe0    str w0, [sp, #28]       XXXXX
    738:    b9401fe0    ldr w0, [sp, #28]       XXXXX
    73c:    910083ff    add sp, sp, #0x20
    740:    d65f03c0    ret
```

Instances of caching

- CPU caches
  - *Why do we have caches on the CPU?* — accessing main memory is ~100x slower than accessing a register
  - Store instructions and data (stack, heap, etc.) from main memory
  - Three levels --- L1, L2, and L3
  - Range in size from a few KB to a few MB
  - Cache line (i.e., cache entry) is typically larger than a word — e.g., 128 bytes
    - *Why?* — spatial locality
  - What happens when we write to memory?
    - Write through cache — write to the cache and main memory
    - Write back cache — initially write to the cache; write to main memory when the entry is evicted from the cache
    - What are the advantages of each approach?
      - Write through cache ensures consistency between CPU cores
      - Write back cache only incurs the overhead of accessing main memory when absolutely necessary
- Web browser caches
  - *Why do web browsers have caches?*
    - Accessing remote network storage is >50x slower than accessing a solid state drive (SSD)
    - Spatial locality — many aspects of a web page are also used with other pages on the same site: e.g., images, Cascading Style Sheets (CSS), JavaScript (JS)
    - Temporal locality — users often visit the same web page repeatedly: e.g., Google
    - Internet Service Provider (ISP) may limit amount of data downloaded/uploaded per month
  - Store static content (e.g., images, CSS, JS)

    - Web browser caches are read-only
- Domain Name System (DNS) caches
    - DNS is used to translate domain names (e.g., `portal.colgate.edu`) into Internet Protocol (IP) addresses (e.g., `149.43.134.29`)
    - DNS entries—i.e., mappings from domain names to IP addresses—can be cached by a web browser, an operating system, and a recursive resolver
        - A recursive resolver is a DNS server within a network that receives DNS queries from clients and queries other DNS servers on the client's behalf in order to locate the desired DNS entry
    - Why are DNS entries cached?
        - Contacting a recursive resolver (and other DNS servers) is much slower than access a solid state drive (SSD)
        - Spatial locality — users often visit subdomains of a domain, e.g., `portal.colgate.edu`, `moodle.colgate.edu`, and `cs.colgate.edu` are subdomains of `colgate.edu`
        - Temporal locality — users often visit the same domain repeatedly
        - Mappings from domain names to IP addresses change infrequently —> DNS entries can be cached for hours or days
- Content distribution networks (CDNs)
    - Collection of geographically distributed servers that delivery content (e.g., streaming videos) to users
    - User's computers contact a server that is "nearby"
        - Ideally measured in terms of latency, which is a function of geographic distance, network routes, and network load
        - Analogy: time it takes to drive somewhere is a function of geographic distance, the route you take, and the amount of traffic on the road
    - CDN servers fetch and cache content from origin servers
    - Popular content (e.g., image from the front page of the NY Times) is more likely to already be cached

Cache replacement

- If a cache is full, then a cache entry must be removed so different data can be placed in the cache
- Cache replacement policy governs which data is removed
- *What should a good cache replacement policy do?* — maximize the number of cache hits (or minimize the number of cache misses)
    - Evaluation metric: Hit ratio = number of hits / total number of memory accesses
- *How do we determine which cache entry to replace?*
- Optimal replacement policy: policy that replaces the entry that will be accessed furthest in the future
    - Impractical because we don’t know data access patterns a priori
- First-in First-out (FIFO)
    - Simple to implement
    - Doesn’t consider the importance of a cache entry
- Random
    - Even simpler to implement
    - Doesn’t consider the importance of a cache entry
- Least Frequently Used (LFU) and Least Recently Used (LRU)
    - Based on the principle of locality
    - LFU assumes a page that is accessed many times will be accessed many more times
    - LRU assumes a page that was accessed recently will be accessed again soon
    - Inverse is very bad replacement policy
    - Downside: lots of overhead to implement — need to store an ordered list of pages and move a page up in the list whenever it’s accessed
    - Where does this go wrong? — when working-set size (i.e., number of repeatedly accessed entries) is (slightly) greater than size of the cache
- *Assume a cache can hold 3 entries and the following 15 data accesses occur: 3, 4, 4, 5, 3, 2, 3, 4, 1, 4, 4, 2, 5, 2, 4. Assuming the cache is initially empty, what is the hit ratio for each of the following algorithms?*
    - *Optimal* — +3, +4, H4, +5, H3, -5/+2, H3, H4, -3/+1, H4, H4, H2, -1/+5, H2, H4 — 9/15 = 60%
    - Q2: *FIFO* — +3, +4, H4, +5, H3, -3/+2, -4/+3, -5/+4, -2/+1, H4, H4, -3/+2, -4/+5, H2, -1/+4 — 5/15 - 33%
    - Q3: *LRU* — +3, +4, H4, +5, H3, -4/+2, H3, -5/+4, -2/+1, H4, H4, -3/+2, -1/+5, H2, H4 — 7/15 = 47%
    - Q4: *LFU* — +3, +4, H4, +5, H3, -5/+2, H3, H4, -2/+1, H4, H4, -1/+2, -2/+5, -5/+2, H4 — 7/15 = 47%