CSCI 210: Computer Architecture
Lecture 36: Caches V

Stephen Checkoway
Slides from Cynthia Taylor
Oberlin College

CS History: IBM System/360 Model 85

Unidentified US government agent, Public domain, via Wikimedia Commons

- First computer to have cached memory
- Shipped in December of 1969
- Had either 16 KB or 32 KB cache
  - Modern systems have around 64 MB
- IBM only built about 30 of them

MAKING CACHES FASTER

Multilevel Caches

- Primary (or level-1) caches (separate instruction and data)
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- L-3 cache usually services multiple CPU cores
- L-3 misses go to main memory

Multilevel Cache performance

- For primary (L-1) cache:
  - Access time is a small number of cycles (less than 10; modern Intel about 4 cycles)
  - Miss rate (fraction of L-1 cache accesses which miss)
  - On a miss, the next level of the cache hierarchy is consulted
- For L-n cache for n > 1:
  - Access time in cycles (10–20 for L2, 30–80 L3)
  - Miss rate (fraction of L-n cache accesses which miss)
  - On a miss, the next level of the cache hierarchy is consulted
- Memory
  - Access time in cycles (on the order of a few hundred cycles)

Cache Example: L-1 only

- Given
  - CPU base CPI = 1
  - L-1 access time = 1 cycle (total, not in addition to the base CPI)
  - Miss rate = 10%
  - Main memory access time = 400 cycles
- With just a primary (L-1) cache
  - Effective CPI = 1 + 0.10 * 400 = 41

Cache example: L-1 and L-2

- L-1:
  - Access time = 1 cycle (so included in the base CPI)
  - Miss rate = 10%
- L-2
  - Access time = 20 cycles
  - Miss rate = 4%
- Memory access time of 400 cycles
- CPI = 1 + 0.10 * (20 + 0.04 * 400) = 4.6
  [Compare to a CPI of 41 for L-1 only]

Cache example: L-1, L-2, L-3

- L-1: access time = 1 cycle; miss rate = 10%
- L-2: access time = 20 cycles; miss rate = 4%
- L-3: access time = 50 cycles; miss rate = 1%
- Memory access time = 400 cycles

With your group, work out what the CPI is assuming a base CPI of 1.

Multilevel Cache Considerations

- Primary cache
  - Focus on minimal hit time
- L-3 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than a single cache
  - L-1 often less associative than L-2 and L-3

Some Actual Numbers: AMD Ryzen 7 5800X

Image from newegg.com

8-core

L1 Cache: 64K per core

L2 Cache: 512K per core

L3 Cache: 32 MB, shared across all cores

AMD sells a “gamer” version with a 96 MB L3 Cache

Launched in November 2020

AMD Ryzen Threadripper PRO 9995WX

96 core (192 threads)

L1 I Cache: 32 kB/core 8-way

L1 D Cache: 48 kB/core 12-way

L2 Cache: 1 MB/core 16-way

L3 Cache: 384 MB shared 16-way

Image from amd.com

Interactions with Advanced CPUs

- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
    - Independent instructions continue

Prefetching

- Hardware Prefetching
  - suppose you are accessing a single field in each object in an array of large objects
  - hardware determines the “stride” and starts grabbing values early
- Software Prefetching
  - Compiler adds extra instructions to load data before it is needed

Which data structure will have better memory access times assuming you have a prefetcher?

A. ArrayList

B. Linked List

C. There will not be any difference

Writing Cache-Aware Code

- Focus on your working set
- If your “working set” fits in L1 it will be vastly better than a “working set” that fits only on disk.
- If you have a large data set – do processing on it in chunks.
- Think about regularity in data structures (can a prefetcher guess where you are going – or are you pointer chasing)

You need to sum every number in multi-dimensional array that is larger than a single cache block. Data is stored so that items in the same row are adjacent in memory. What code should you use to sum it?

```
sum = 0
for i in range(0, num_cols):
    for j in range(0, num_rows):
        sum +=  arr[j][i]
```

A

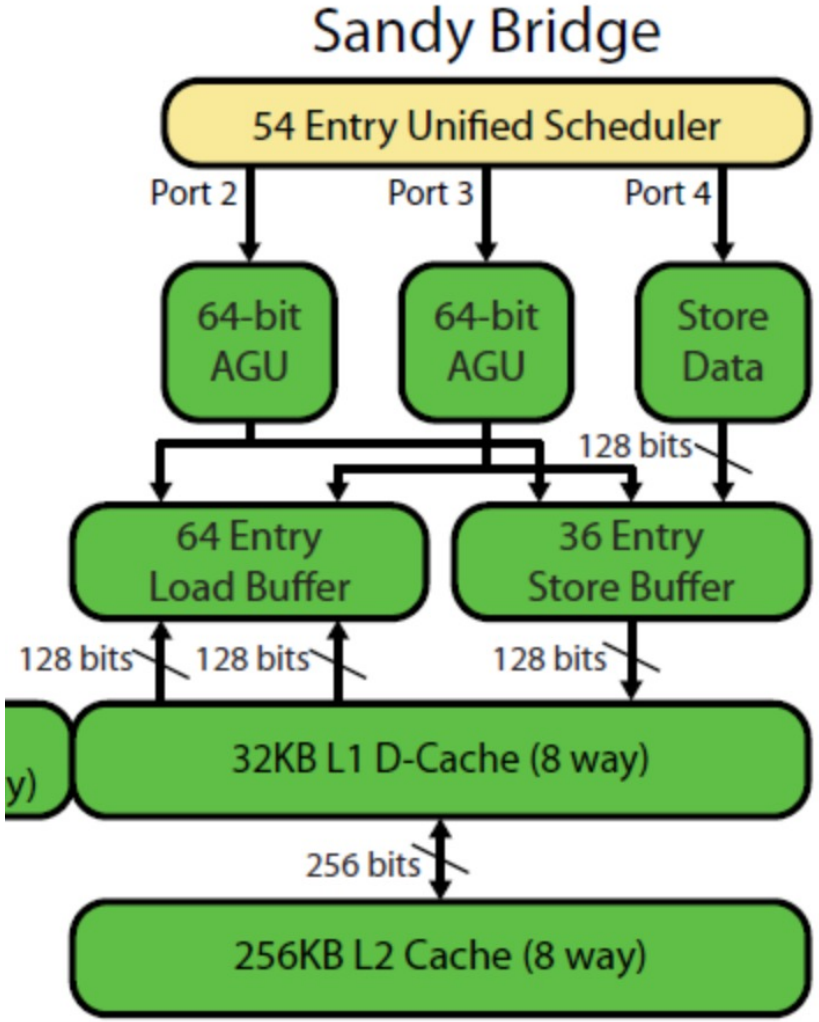
```
sum = 0
for i in range(0, num_rows):
    for j in range(0, num_cols):
        sum+= arr[i][j]
```

B

C. They both have the same performance

Real world is slightly different than presented

- L1 cache access is typically > 1 cycle
- Caches are “multi-port” meaning they can perform more than one operation per cycle
- Out-of-order execution
- Virtual memory (virtually indexed, physically tagged)
- Sandy Bridge is now old (~2010) but has
  - Two 128-bit loads and one 128-bit store per cycle (throughput, not latency)
  - 64 load µops, 36 store µops “in flight” at a time
  - 32 kB, 8-way set associative L1 data cache
  - 256 kB 8-way set associative L2 data cache



https://www.realworldtech.com/sandy-bridge/7/

CACHE SIMULATOR PROJECT

Two parts

1. Cache simulator
2. Use the simulator to derive a good cache configuration and answer questions about it

The two parts are equally weighted

Part 1: Cache Simulator

- Take in a data trace of load/stores from a real program
- Simulate running the program on a given cache
- Calculate how well a given cache would perform for that trace

Cache Parameters

- Always: Write-allocate, write-back, LRU replacement

- Change:
  - Cache size
  - Block size
  - Associativity
  - Hit time
  - Miss penalty

Address Trace

Load/Store Address InstructionCount

```
# 0 7fffed80 1
# 0 10010000 10
# 0 10010060 3
# 1 10010030 4
# 0 10010004 6
# 0 10010064 3
# 1 10010034 4
```

L/S: 0 for load, 1 for store

Simulation Results

```
Simulation results:
    execution time                 52268708 cycles
    instructions                    5136716
    memory accesses                 1957764
    overall miss rate                  0.79
    load miss rate                     0.88
    CPI                               10.18
    average memory access time        24.07 cycles
    dirty evictions                  225876
    load_misses                     1525974
    store_misses                      30034
    load_hits                        205909
    store_hits                       195847
```

What do you need to do?

- Create data structures that emulate a cache
- For each memory access, find where it would go in the cache, check if it’s already there
- Calculate number of miss penalty cycles, load misses, store misses, instructions, etc.

Part 2: Choosing good parameters

- You're going to select cache size, associativity, and block size by looking at real program memory accesses
- Three programs: art, mcf, and swim from the SPEC benchmarks
- For each parameter, you're going to compare the performance of 3 settings on each of the 3 programs
- From these 9 results, you'll select the best value of the parameter and then move on to the next parameter

Difficulties

- Increasing cache size/associativity increases the cache hit time
- Increasing the block size increases the cache miss penalty
- For each parameter, there isn't necessarily a single best choice
  - E.g., increasing the associativity may help some programs but not all of them
- Select the choice that works best over all
- After performing the 27 simulations, the final choice may actually be worse than the baseline cache configuration on one of the programs!

Making a parameter choice

- One way to choose a parameter:
  1. Run the 3 simulations with a given parameter choice
  2. Compute the speedup of each relative to the baseline
  3. Take the average of the speedups across the 3 simulations
  4. Select the parameter value with the highest average speedup

- Be careful though! The arithmetic mean of the speedups is **meaningless!** You must use the geometric mean (see the reading)

Answering the questions

- The main output of the second part of the project is answers to several questions
- You will have performed 27 simulations and you should have a lot of data at this point
- You should explicitly use this data to answer the questions!
- Graphs or tables can be helpful

Reading

- Fleming and Wallace “How not to lie with statistics: the correct way to summarize benchmark results” https://dl.acm.org/doi/10.1145/5666.5673