Prefetching

Advanced Topics in Computer Architecture

Timothy Jones
Caching

• We’re all familiar with caching
• Caches store data close to the core
• Caches take advantage of locality
  • Spatial locality
  • Temporal locality
Cache performance

• Cache hit and miss rates give an indication of cache performance
  • But they fail to capture the impact of the cache on the overall system
• We therefore prefer to incorporate timing into the cache performance
  • For example, including the time take to access the cache
  • And the time taken to service a miss
• This can give us a value for the average memory access time (AMAT)
Characterising cache performance

• From the CPU’s point of view, we want to reduce the average memory access time (AMAT)
  • This is the average time it takes to load data
  • Including a cache in the system should lead to reducing AMAT, otherwise it is doing more harm than good!

\[
\text{AMAT} = \text{Cache hit time} + \text{Cache miss rate} \times \text{Cache miss penalty}
\]
Improving cache performance

\[ \text{AMAT} = \text{Cache hit time} + \text{Cache miss rate} \times \text{Cache miss penalty} \]

• Let’s consider the equation further to see how to reduce AMAT
• We can’t improve the cache hit time, this is fixed
• The cache miss penalty depends on where else the data is
  • I.e. whether it is in other caches or main memory
  • The AMAT of that cache dictates this!
• We have the most control over the cache miss rate
  • We can classify cache misses into four categories
Classifying cache misses

Compulsory misses

• These occur when the data at the memory location being accessed has never existing in the cache

• The first access to any new block generates a compulsory miss

Diagram:

- Cache
- Main memory
Classifying cache misses

Compulsory misses

• These occur when the data at the memory location being accessed has never existing in the cache
• The first access to any new block generates a compulsory miss
Classifying cache misses

Conflict misses

• When too many memory locations map to the same set, some blocks have to be evicted and reloaded; this generates conflict misses

• Conflict misses only occur in direct-mapped and set-associative caches
Classifying cache misses

**Conflict misses**

- When too many memory locations map to the same set, some blocks have to be evicted and reloaded; this generates conflict misses
- Conflict misses only occur in direct-mapped and set-associative caches
Classifying cache misses

Conflict misses

• When too many memory locations map to the same set, some blocks have to be evicted and reloaded; this generates conflict misses

• Conflict misses only occur in direct-mapped and set-associative caches
Classifying cache misses

Capacity misses

• When there is not enough space in the cache to hold all the data required, some of it must be evicted and reloaded when next accessed

• In other words, the cache simply could not hold all of the data required at once
Classifying cache misses

**Capacity misses**

- When there is not enough space in the cache to hold all the data required, some of it must be evicted and reloaded when next accessed.
- In other words, the cache simply could not hold all of the data required at once.
Classifying cache misses

Capacity misses

• When there is not enough space in the cache to hold all the data required, some of it must be evicted and reloaded when next accessed
• In other words, the cache simply could not hold all of the data required at once
Classifying cache misses

Coherence misses

• If there is a cache coherence protocol running then when one core attempts to write to some data, the protocol invalidates that address in another cache

• Reloading that data in that other cache is a coherence miss – this wouldn’t occur without the coherence protocol
Classifying cache misses

Coherence misses

• If there is a cache coherence protocol running then when one core attempts to write to some data, the protocol invalidates that address in another cache

• Reloading that data in that other cache is a coherence miss – this wouldn’t occur without the coherence protocol
Classifying cache misses

Coherence misses

• If there is a cache coherence protocol running then when one core attempts to write to some data, the protocol invalidates that address in another cache

• Reloading that data in that other cache is a coherence miss – this wouldn’t occur without the coherence protocol
Classifying cache misses

Coherence misses

- If there is a cache coherence protocol running then when one core attempts to write to some data, the protocol invalidates that address in another cache.
- Reloading that data in that other cache is a coherence miss – this wouldn’t occur without the coherence protocol.
Reducing cache misses

• We can reduce the number of misses in some of these classes directly
• For example, conflict misses
  • These can be reduced by increasing the size of each set
• Or capacity misses
  • These could be reduced by increasing the size of the cache
• However, we’re going to focus here on schemes to improve all misses
  • All schemes employ some notion of *prefetching*
Prefetching

• This is a technique to bring data into the cache before it is needed
• The idea is to make a prediction about what data the program will use in the near future
  • Then load that data into the cache so that it arrives *before* required
• Prefetching can be performed in hardware or software
  • Processors often provide special instructions to do this in software
• We’re going to look at a variety of hardware techniques
A simple prefetcher

• Next-line is a simple prefetcher
  • Does what it says on the tin!

• Stride prefetchers are also relatively simple

• The prefetcher identifies simple patterns in the accesses made
  • E.g. 0x1000, 0x1100, 0x1200

• It learns this stride and prefetches based on it
A simple prefetcher

- Next-line is a simple prefetcher
  - Does what it says on the tin!
- Stride prefetchers are also relatively simple
- The prefetcher identifies simple patterns in the accesses made
  - E.g. 0x1000, 0x1100, 0x1200
- It learns this stride and prefetches based on it
A simple prefetcher

- Next-line is a simple prefetcher
  - Does what it says on the tin!
- Stride prefetchers are also relatively simple
- The prefetcher identifies simple patterns in the accesses made
  - E.g. 0x1000, 0x1100, 0x1200
- It learns this stride and prefetches based on it
More complex prefetching

• Stride prefetchers are effective for a lot of workloads
  • Think array traversals
• But they can’t pick up more complex patterns
• In particular two types of access pattern are problematic
  • Those based on pointer chasing
  • Those that are dependent on the value of the data
• More complex prefetchers are required for this
Prefetching questions

• Whilst reading the papers for next week, here are some questions you might like to think about to judge each approach

• How do the prefetchers make their predictions?
  • Does this have a bearing on the access patterns that can be prefetched?

• What are the hardware requirements of the schemes?
  • I.e. what structures are needed to implement it and how costly are they?

• Where does the data get prefetched to?
  • Most of the time you’d like it brought into your own L1 cache

• What is the impact on other parts of the system (core, caches, etc)?