How to Build a Cache
COMP 252 - Lecture 2

Antoniu Pop

antoniu.pop@manchester.ac.uk
Computing technology trends, challenges & [some] solutions

- Rapid increase of processing power
- Slow increase of memory access capability
- Memory wall
- Solutions: Caches, out-of-order execution, prefetching, compiler optimizations, etc.
- No silver bullet...

Core concepts behind caches

- What: small, fast memory – close to the CPU
- Why: Speed imbalance – accessing data can take 10s-100s of cycles
- How: Spatial & Temporal Locality
Cache Labs: Objectives

Test:
- Cache behaviour
- Parameters
- Limits

Learn:
- How the cache works
- How memory accesses are handled
- What inhibits the cache’s behaviour

Understand:
- Importance of caches
- How to avoid inefficient patterns
- How to write, optimize programs to take advantage of caches
Today’s Lecture – Learning Objectives

To understand

- how cache is logically structured
- how cache operates
  - CPU reads
  - CPU writes
- key cache performance parameters
- making cache more efficient
Cache Functionality

- The CPU gives the cache a full (e.g. 32 bit) address to access some data
- The cache contains a (small) selection of values from main memory
  - Each such value in cache comes from a particular main memory location
- How do we find the data?
- It may or may not be in the cache
Fully Associative Cache (1)

- Cache is small
- Only holds a few memory values (32k ?)
- Cache stores both addresses and data

Hardware compares input address with all stored addresses
Parallel lookup of (key/value) pairs
Fully Associative Cache (2)

- If address is found
  - this is a “cache hit”
  - data is read (or written)
  - no need to access main RAM memory – fast

- If address is not found
  - this is a “cache miss”
  - must go to main RAM memory – slow

- But how does data get into the cache?

- If we have to go to main memory, should we just leave the cache as it was?
Locality

Caches rely on locality

- **Temporal Locality**
  - things when used will be (likely) used again soon
  - e.g. instructions and data in loops

- **Spatial locality**
  - things close together (adjacent addresses) in store are often used together
  - e.g. instructions or arrays
Cache Hit Rate

- Fraction of cache accesses which “hit” in cache
- Need $\geq 98\%$ to hide $50 : 1$ ratio of memory speed to instruction speed
- Hit rates for instructions usually better than for data
  - more regular access patterns
  - more locality
What to do on a Cache Miss

- Temporal locality says it is a good idea to put recently used data in the cache

- Memory reads
  - Put newly read value into the cache (just the value required?)
  - But cache may be already full
  - Need to choose a location to reject (replacement policy)
Cache Replacement Policy

- Least Recently Used (LRU)
  - makes sense
  - but expensive to implement (in hardware)

- Round Robin (or cyclic)
  - cycle round locations
  - least recently fetched from memory

- Random
  - easy to implement
  - not as bad as it might seem
Cache Write Strategy

- Memory Writes are slightly more complex than reads
- Must do an address comparison first to see if address is in cache
- Cache hit
  - Update value in cache
  - But what about value in RAM?
  - If we write to RAM every time it will be slow
  - Does RAM need updating every time?
Cache Hit Write Strategy

- **Write Through**
  - Every cache write is also done to memory

- **Write Through with buffers**
  - Write through is buffered i.e processor doesn’t wait for it to finish (but multiple writes could back up)

- **Copy Back**
  - Write is only done to cache (mark as “dirty”)
  - RAM is updated when “dirty” cache entry is replaced (or cache is flushed e.g. on process switch)
Cache Miss Write Strategy

Cache miss on write

▶ Write Allocate
  ▶ Find a location (reject an entry if necessary)
  ▶ Assign cache location and write value
  ▶ Write through back to RAM
  ▶ Or rely on copy back later

▶ Write Around (or Write no-Allocate)
  ▶ Just write to RAM
  ▶ Subsequent read will cache it if necessary
Overall Cache Write Strategy

- Fastest is Write Allocate / Copy Back
- But cache & main memory are not “coherent”
  - Can have different values until the cache line is written back
- Does this matter?
  - Autonomous I/O devices
  - Exceptions
  - Multi-processors
- May need special handling (later)
Cache Data Width

Each cache line is:

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
</table>

- If address is e.g. 32 bit byte address, logically data is one byte
- In practice we hold more data per address
  - e.g a 4 byte (32 bit) word
  - stored address needed is then only 30 bits
- In real practice (later) we would hold even more data per address
- Efficiency?
  - How many bits are effectively used to store data?
Real Cache Implementations

- Fully associative cache is logically what we want
  - But expensive!
  - Need to store full (e.g., 32 bit) addresses
  - Hardware comparison is expensive (logic) and uses a lot of power

- It is possible to achieve most of the functionality using ordinary small and fast (usually static) RAM memory in special ways

- Few real processor caches are fully associative
Direct Mapped Cache (1)

- Uses standard RAM to implement the functionality of an ideal cache (with some limitations)

- Usually uses static RAM – although more expensive than DRAM, is faster

- But overall is considerably cheaper to implement than fully associative
Direct Mapped Cache (2)

Address divided into two parts

- index (e.g., 15 bits) used to address (32k) RAMs directly
- Rest of address, the tag, is stored and compared with incoming tag

- If same (hit) data is read
- If different (miss) data is not used
Direct Mapped Cache (3)

- Logically, the RAM is divided in blocks of size equal to cache size
- Each memory location has an assigned cache entry
Direct Mapped Cache (4)

- Is really just a hash table implemented in hardware
- Index / Tag could be selected from address in many ways
- Most other aspects of operation are identical to fully associative
- Except for replacement policy
Direct Mapped Replacement

- New incoming tag/data can only be stored in one place - dictated by index

- Using LSBs as index exploits principle of spatial locality to minimize displacing recently used data

- Much cheaper than associative (and faster?) but lower “hit rate” (due to inflexible replacement strategy)
Set Associative Caches (1)

▶ A compromise!

▶ A set associative cache is simply a small number (e.g. 2 or 4) of direct mapped caches operating in parallel

▶ If one matches, we have a hit and select the appropriate data.
E.g., 2-way set associative cache
Set Associative Caches (2)

- Now replacement strategy can be a bit more flexible

- E.g. in a 4 way, we can choose any one of 4 - using LRU, cyclic etc.

- Now (e.g.) 4 entries all with the same index but different tag can be stored. Less “interference” than direct mapped.
Set Associative Caches (3)

► Hit rate gets better with increasing number of ways

► Obviously higher no. of ways gets more expensive

► In fact N location fully associative cache is the same as a N way set associative cache!
### Example

<table>
<thead>
<tr>
<th>Cache 0</th>
<th>Cache 1</th>
<th>RAM</th>
<th>Cache 0</th>
<th>Cache 1</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>000</td>
<td>A</td>
<td>B</td>
<td>000</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>001</td>
<td>C</td>
<td>D</td>
<td>001</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>100</td>
<td>E</td>
<td>F</td>
<td>100</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
<td>110</td>
<td>G</td>
<td>H</td>
<td>110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RAM</th>
<th>Cache 0</th>
<th>Cache 1</th>
<th>RAM</th>
<th>Cache 0</th>
<th>Cache 1</th>
<th>RAM</th>
<th>Cache 0</th>
<th>Cache 1</th>
<th>RAM</th>
<th>Cache 0</th>
<th>Cache 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>A</td>
<td>B</td>
<td>00001</td>
<td>A</td>
<td>B</td>
<td>00010</td>
<td>A</td>
<td>B</td>
<td>00011</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>00010</td>
<td>C</td>
<td>D</td>
<td>00100</td>
<td>C</td>
<td>D</td>
<td>00110</td>
<td>C</td>
<td>D</td>
<td>00111</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>00100</td>
<td>E</td>
<td>F</td>
<td>01000</td>
<td>E</td>
<td>F</td>
<td>01010</td>
<td>E</td>
<td>F</td>
<td>01011</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>00110</td>
<td>G</td>
<td>H</td>
<td>01100</td>
<td>G</td>
<td>H</td>
<td>01110</td>
<td>G</td>
<td>H</td>
<td>01111</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>00111</td>
<td></td>
<td></td>
<td>01110</td>
<td></td>
<td></td>
<td>01111</td>
<td></td>
<td></td>
<td>01111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td></td>
<td></td>
<td>01100</td>
<td></td>
<td></td>
<td>01110</td>
<td></td>
<td></td>
<td>01111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td></td>
<td></td>
<td>01110</td>
<td></td>
<td></td>
<td>01111</td>
<td></td>
<td></td>
<td>01111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
<td>01111</td>
<td></td>
<td></td>
<td>01111</td>
<td></td>
<td></td>
<td>01111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td>01100</td>
<td></td>
<td></td>
<td>01100</td>
<td></td>
<td></td>
<td>01100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td></td>
<td></td>
<td>01110</td>
<td></td>
<td></td>
<td>01110</td>
<td></td>
<td></td>
<td>01110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td></td>
<td></td>
<td>01111</td>
<td></td>
<td></td>
<td>01111</td>
<td></td>
<td></td>
<td>01111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>