



EPSRC

Introduction to OpenMP

Lecture 10: Caches



| epcc |



Overview

- Why caches are needed
- How caches work
- Cache design and performance.



The memory speed gap

- Moore's Law: processors speed doubles every 18 months.
 - True for last 35 years....
- Memory speeds (DRAM) are not keeping up (double every 5 years) .
- In 1980, both CPU and memory cycles times were around 1 microsecond.
 - Floating point add and memory load took about the same time.
- In 2000 CPU cycles times were around 1 nanosecond, memory cycle times around 100 nanoseconds.
 - Memory load is 2 orders of magnitude more expensive than floating point add.



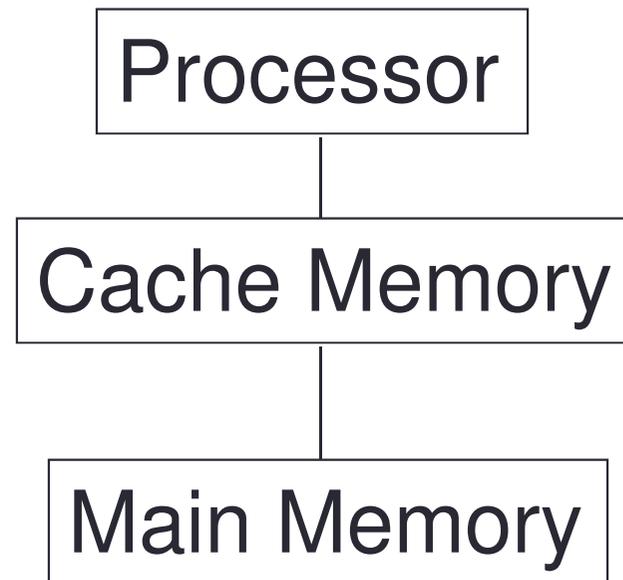
Principal of locality

- Almost every program exhibits some degree of locality.
 - Tend to reuse recently accessed data and instructions.
- Two types of data locality:
 1. Temporal locality
 - A recently accessed item is likely to be reused in the near future.
e.g. if x is read now, it is likely to be read again, or written, soon.
 2. Spatial locality
 - Items with nearby addresses tend to be accessed close together in time.
e.g. if $y[i]$ is read now, $y[i+1]$ is likely to be read soon.



What is cache memory?

- Small, fast, memory.
- Placed between processor and main memory.



How does this help?

- Cache can hold copies of data from main memory locations.
- Can also hold copies of instructions.
- Cache can hold recently accessed data items for fast re-access.
- Fetching an item from cache is much quicker than fetching from main memory.
 - 1 nanosecond instead of 100.
- For cost and speed reasons, cache is much smaller than main memory.



Blocks

- A cache **block** is the minimum unit of data which can be determined to be present in or absent from the cache.
- Normally a few words long: typically 32 to 128 bytes.
- See later for discussion of optimal block size.
- N.B. a block is sometimes also called a **line**.



Design decisions

- When should a copy of an item be made in the cache?
- Where is a block placed in the cache?
- How is a block found in the cache?
- Which block is replaced after a miss?
- What happens on writes?

Methods must be **simple** (hence cheap and fast to implement in hardware).



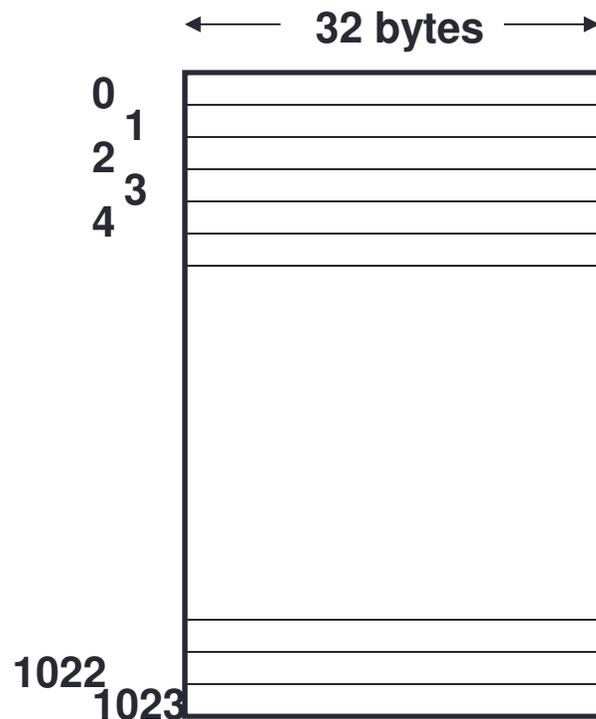
When to cache?

- Always cache on reads
 - except in special circumstances
- If a memory location is read and there isn't a copy in the cache (**read miss**), then cache the data.
- What happens on writes depends on the write strategy: see later.
- N.B. for instruction caches, there are no writes



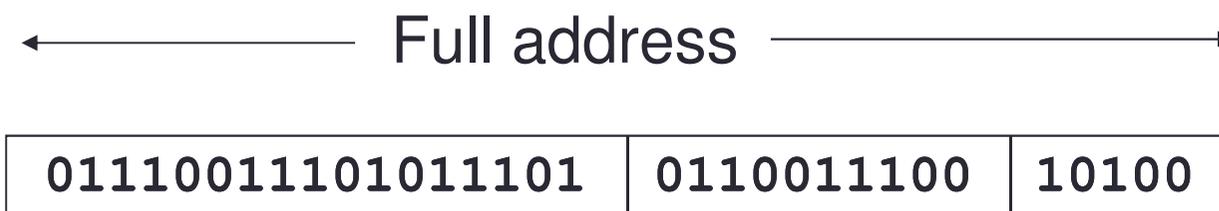
Where to cache?

- Cache is organised in **blocks**.
- Each block has a number:



Bit selection

- Simplest scheme is a **direct mapped** cache
- If we want to cache the contents of an address, we ignore the last n bits where 2^n is the block size.
- Block number (index) is:
(remaining bits) MOD (no. of blocks in cache)
 - next m bits where 2^m is number of blocks.



block
index

block
offset



Set associativity

- Cache is divided into sets
- A set is a group of blocks (typically 2 or 4)
- Compute set index as:
(remaining bits) MOD (no. of sets in cache)
- Data can go into any block in the set.



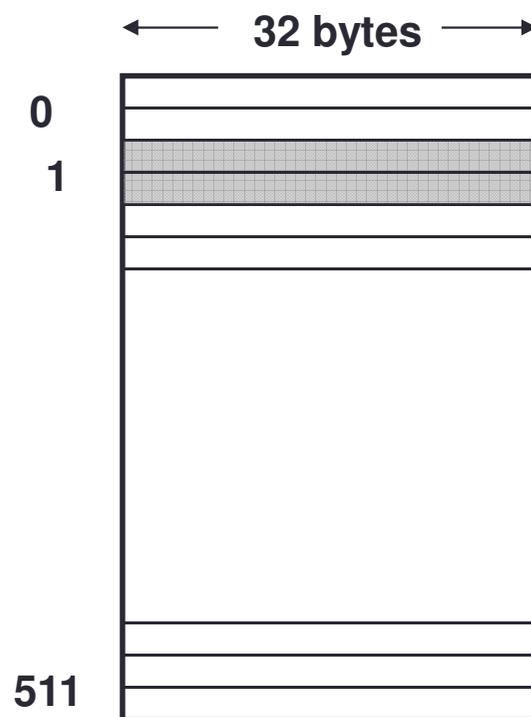
set
index

block
offset



Set associativity

- If there are k blocks in a set, the cache is said to be k -way set associative.



- If there is just one set, the cache is **fully associative**.



How to find a cache block

- Whenever we load an address, we have to check whether it is cached.
- For a given address, find set where it might be cached.
- Each block has an address tag.
 - address with the block index and block offset stripped off.
- Each block has a valid bit.
 - if the bit is set, the block contains a valid address
- Need to check tags of all valid blocks in set for target address.



tag

set
index

block
offset



Which block to replace?

- In a direct mapped cache there is no choice: replace the selected block.
- In set associative caches, two common strategies:

Random

- Replace a block in the selected set at random.

Least recently used (LRU)

- Replace the block in set which was unused for longest time.
- LRU is better, but harder to implement.



What happens on write?

- Writes are less common than reads.
- Two basic strategies:

Write through

- Write data to cache block and to main memory.
- Normally do not cache on miss.

Write back

- Write data to cache block only. Copy data back to main memory only when block is replaced.
- Dirty/clean bit used to indicate when this is necessary.
- Normally cache on miss.



Write through vs. write back

- With write back, not all writes go to main memory.
 - reduces memory bandwidth.
 - harder to implement than write through.
- With write through, main memory always has valid copy.
 - useful for I/O and for some implementations of multiprocessor cache coherency.
 - can avoid CPU waiting for writes to complete by use of write buffer.



Cache performance

- Average memory access cost =

$$\text{hit time} + \text{miss ratio} \times \text{miss time}$$

time to load data
from cache to CPU

proportion of accesses
which cause a miss

time to load data from
main memory to cache

- Can try to to minimise all three components



Cache misses: the 3 Cs

- Cache misses can be divided into 3 categories:

Compulsory or cold start

- first ever access to a block causes a miss

Capacity

- misses caused because the cache is not large enough to hold all data

Conflict

- misses caused by too many blocks mapping to same set.



Block size

- Choice of block size is a tradeoff.
- Large blocks result in fewer misses because they exploit spatial locality.
- However, if the blocks are too large, they can cause additional capacity/conflict misses (for the same total cache size).
- Larger blocks have higher miss times (take longer to load)



Set associativity

- Having more sets reduces the number of conflict misses.
 - 8-way set associate is almost as good as fully associative.
- Having more sets increases the hit time.
 - takes longer to find the correct block.
- Conflict misses can also be reduced by using a **victim cache**
 - a small buffer which stores the most recently evicted blocks
 - helps prevent thrashing, where subsequent accesses all resolve to the same set.



Prefetching

- One way to reduce miss rate is to load data into cache before the load is issued. This is called **prefetching**
- Requires modifications to the processor
 - must be able to support multiple outstanding cache misses.
 - additional hardware is required to keep track of the outstanding prefetches
 - number of outstanding misses is limited (e.g. 4 or 8): extra benefit from allowing more does not justify the hardware cost.

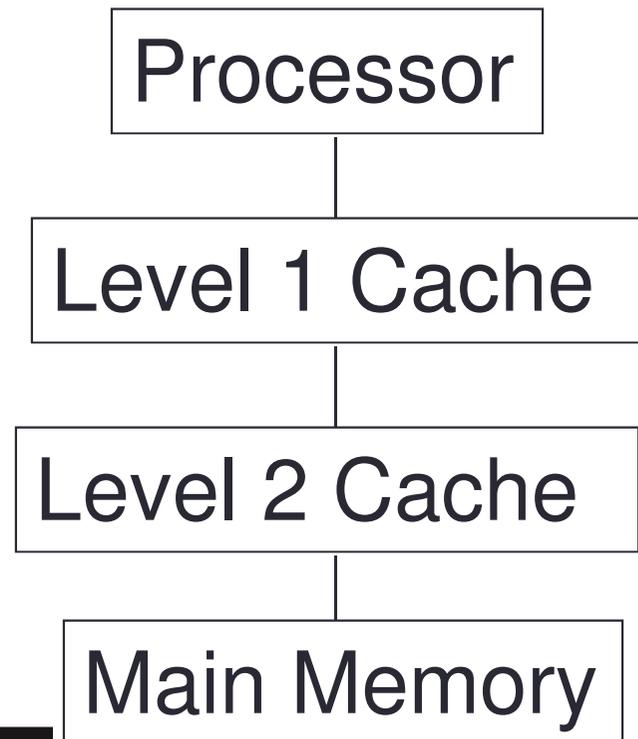


- Hardware prefetching is typically very simple: e.g. whenever a block is loaded, fetch consecutive block.
 - very effective for instruction cache
 - less so for data caches, but can have multiple streams.
 - requires regular data access patterns.
- Compiler can place prefetch instructions ahead of loads.
 - requires extensions to the instruction set
 - cost in additional instructions.
 - no use if placed too far ahead: prefetched block may be replaced before it is used.



Multiple levels of cache

- One way to reduce the miss time is to have more than one level of cache.



Multiple levels of cache

- Second level cache should be much larger than first level.
 - otherwise a level 1 miss will almost always be level 2 miss as well.
- Second level cache will therefore be slower
 - still much faster than main memory.
- Block size can be bigger, too
 - lower risk of conflict misses.
- Typically, everything in level 1 must be in level 2 as well (**inclusion**)
 - required for cache coherency in multiprocessor systems.



Multiple levels of cache

- Three levels of cache are now commonplace.
 - All 3 levels now on chip
 - Common to have separate level 1 caches for instructions and data, and combined level 2 and 3 caches for both
- Complicates design issues
 - need to design each level with knowledge of the others
 - inclusion with differing block sizes
 - coherency....



Memory hierarchy

