A Tutorial on Processor Caches

Noah Mendelsohn
Tufts University
Email: noah@cs.tufts.edu
Web: http://www.cs.tufts.edu/~noah
This slide deck is intended as self-study material...

...if you have PowerPoint, then run it as a “Slide Show”...

Windows: press F5
Mac: AppleKey+RETURN
Goals for this presentation

- Explain how caches make memories faster without adding too much cost
- Explain how memory references are resolved using caches
- Explain different cache organizations:
  - Direct mapped, fully-associative, set associative
- Explain how the bits in an address are used to select data from the cache
- Explain how to analyze the performance of a cache
The material in this presentation follows a progression…

…first we consider a machine with no cache at all…

…then we go through progressively more complicated designs until we explore an n-way set-associative design that’s typical of modern systems…

…along the way, we talk about how address bits select lines in a cache, and at the end, how to analyze cache performance.
First, consider a system with no cache...
CPU Reads data from memory

Request contents of block at address 320 (hex 140)

CPU

Memory

XYZ

0 64 128 192

256

512

768

1024

1280

1536

1792

2048
CPU Reads data from memory

Request contents of block at address 320 (hex 140)

Note: address 320 (hex 140) is here

CPU

Memory
CPU **Writes** data to memory

Write “ABC” to block at address 128 (hex 80)
CPU **Writes** data to memory

Write “ABC” to block at address 128 (hex 80)
Summary of CPU Access to Memory

- **Read:**
  - CPU presents address to memory
  - *Aligned* block of data (e.g. 64 bytes) from that address comes back
  - CPU will typically extract specific bytes needed (e.g. 4 byte integer)

- **Write**
  - CPU presents address and new data to memory
  - Memory updates
  - (In practice, data is updated a block at a time, but that’s usually done with help of the cache, which we’ll see next…without the cache, the CPU might have to read a whole block first, update part of it, and write it back).

- **Remember:** the memory updates in *aligned blocks*, typically 64 or maybe 128 bytes
What’s the problem?

- Big memories are either very expensive or very slow
  - Typically, we build relatively cheap memories that may take ~ 100 instruction times to read a block
- We can build much faster memories, but the cost per byte is high
- The trick: build a small fast cache of expensive memory, and try to automatically keep the most frequently accessed data there
The simplest possible cache…
…just one “line”
CPU Reads data from memory

Request contents of block at address 320 (hex 140)

Cache
CPU Reads data from memory

Request contents of block at address 320 (hex 140)
CPU Reads data from memory

Request contents of block at address 320 (hex 140)

Data flows from memory to cache, and then to the CPU
CPU Reads data from memory

If data from the block is requested again, we can skip reading the slow memory.

Request contents of block at address 320 (hex 140).

Data flows directly from the cache – *no memory access*!!
NOTE: The cache provides the same service that the memory itself did...give it an address, it gives back the data
CPU Reads data from memory

Ooops...we also need to remember *which* block of data is in the cache

**CPU**

**Memory**

- 64
- 128
- 192
- 256
- 512
- 768
- 1024
- 1280
- 1536
- 1792
- 2048

**Cache**
CPU Reads data from memory

Each line has a tag which is the address of the first byte cached
CPU Reads data from memory

Data flows from memory to cache, and then to the CPU

The part of the *line* that holds the data is called the *block*.
CPU Reads data from memory

Data flows from memory to cache, and then to the CPU
Data flows from memory to cache, and then to the CPU.
Before we look at more complicated caches…

…let’s change to numbering our addresses in hex, and look a bit more closely at the “tag”
These are the same addresses, but in hex:

Remember:

320 (dec) = 140 (hex) = 0000000101000000 (bin)
The earlier slides lie about what’s in the tag

The cache line holds 64 bytes...

...we said that the tag would hold the address of the first byte, but the low order 6 bits of the address of the first byte in any block are always zero...so why waste space holding 6 bits that are always zero?

Sincere apologies for the tacky animation, but I wanted to get your attention!
Let’s look in binary:

140 (hex) = 0000000101000000

We don’t need tag bits for the positions that are always zero, so the tag really contains 000000101 (binary) = 5 (hex).

Convince yourself that this is because the cache happens to hold the 6th memory block! (I.e. block 5 starting from block 0)
What have we learned so far?

- We’ve seen how a simple 1 line cache can improve performance by keeping recent data in faster memory.
- We’ve seen how the tag remembers enough of the address for us to know which block of memory is in the cache line.
- We’ve seen how writing addresses in hex (or binary) makes it easier to keep track of which bits matter:
  - In hex, powers of two are easy to spot, because the low order bits are zero.
  - Since memories are typically organized into blocks with sizes that are powers of two, numbering in hex is convenient.
  - When we want to see exactly which bits are needed for tags, we’ll convert to binary (and that’s trivial if we start with hex).
If we had a two line cache, then we could cache two things at a time.
We’re going to see three different approaches to cache design: fully associative, direct mapped, and n-way set associative.
Two line cache

Question: can any block of data be cached in any cache line?
Rarely, we build caches that are fully associative...in these, any block of data can be stored in any line.

Problem: large fully associative caches are hard to engineer, because you might have to check hundreds of lines to see if a given address has been cached.
Fully associative caches are complicated to build, because you have to implement the hardware for parallel comparisons.

All address bits except those for selecting within a block are in the tag.

Suitable for smaller caches.
Direct mapped caches

To avoid that cost, we can build *direct mapped* caches...each block in memory has only one cache line where it can go.

Block 5 (address 0x140) can be cached *only* in line 1 of the cache, because it’s an odd numbered block....(using zero-based numbering)

Now the hardware needs to check only one line for each request
To avoid that cost, we can build direct mapped caches...each block in memory has only one cache line where it can go.

In general, mapping just cycles through the blocks, according to the number of cache lines...which we’ve increased here to four.

Each memory block has only one possible cache line.
Let’s look at tags again – we were lying again!

140 (hex) = \text{0000000101000000}

We decided before that we could avoid storing the low order 6 bits that identify bytes \textit{in} the block.

With a direct mapped cache, we also don’t need to store the \textbf{bits that identify the line}, which in this case is the red pair of bits \texttt{01}.

\texttt{0000000101} (binary) = 5 \ (hex).
\texttt{00000001} (binary) = 1 \ (hex).

Indeed, we’re storing the 2\textsuperscript{nd} block that can go in the cache line for red blocks, and since we number from zero, that’s numbered “1”. We need not store the low order 8 bits in the tag. \textit{If there were eight lines, we would not have to store the low order 9 bits. Make sure you understand why!}
Almost done….

The fully associative cache was great, because any data could be cached in any line…but we couldn’t build a big one.

Direct map solved that, but now we can’t cache at the same time two lines from the same “color” group in memory!
A tricky compromise:
n-way set associative caches have more than one line for each mapping group...

within a group, access is associative
4-way set associative with 8 lines

Just an example...

A typical cache might have hundreds of sets, but usually not more than 8- or 16-way associative within the sets.

Block 5 (address 0x140) can be cached *in any of the 4 lines that make up set 1*...there’s now some lookup cost for checking those 4 lines on access to any odd block.
Question: What about those pesky tag bits?

What tag information do we need to store in the above example?

- Blocks are still 64 bites, so we don’t need to store the low 6 bits
- There are two sets, so the next address bit will be used to determine which set the block will go in. That bit also won’t be part of the tag (but will be used to select the set).
- That is, there is a 1-bit set index.
- Note that any line in the set can hold any block of a particular color (any even numbered block for set 0, any odd numbered block for set 1).
- All the bits but the low 7 will be tag bits. So, our block with address 0x140, will have a tag of 0x2 = 10 (in binary)
- If addresses were 16 bits, then 16 – 7 = 9 bits would be used as the tag
And there is another bit that isn’t part of the address

How does the cache know whether a cache line contains data loaded from memory (as opposed to dead squirrels)?
And there is another bit that isn’t part of the address

How does the cache no whether a cache line contains data loaded from memory (as opposed to dead squirrels)?

The cache keeps a **valid bit** for each cache line. This bit is a *secret of the cache’s*: it’s not visible to the CPU or to memory; it isn’t part of the address; and it isn’t part of the data. It’s for the cache’s own bookkeeping.

When the machine starts up, the cache has no data in it, so the valid bit for each line is set to 0 initially. When data is read from the memory, it’s copied into an appropriate cache line, and the valid bit is set to 1.

This allows the cache to evict a block without having to store new data in the corresponding cache line. It just sets the valid bit to 0. We haven’t seen why someone might want this.
Note that for any cache or set that's associative, the system must have a *replacement policy*... when new data is cached, we have to figure out which existing data to *evict*.

Hardware often tries to approximate *Least Recently Used (LRU)*... that is, throw out the block which has been least recently accessed.
Handling Writes
Two approaches to handling writes

- **write-through** (sometimes called store-through)
  - Write to memory on each update
  - Easier to implement
  - Causes lots of slow write traffic when the same block is updated repeatedly

- **write-back or store-in**
  - Update only the cache…don’t update memory until eviction time
  - Needs a *dirty bit* for each line to note whether there are unwritten updates (another bit that a secret of the cache)
  - May make evictions slower
  - Usually better performance, unless repeated updates to a block are rare
Summary of Cache Architecture
Summary

- Caches attempt to put frequently used data in high speed memory that’s easily accessed by the CPU

- A fully associative cache allows maximum flexibility
  - Any data can go in any line
  - Usually too expensive/complex/impractical to implement

- A direct mapped cache is simple, but offers no placement flexibility

- Set associative caches are a middle ground compromise

- Tags record which block of memory is in each line
  - Only bits needed to distinguish that data from other blocks that could be stored in that line are explicitly recorded in the tag
Putting it Together

Resolving a 64 Bit AMD64 address in a realistic cache
A more realistic cache

Block size: 64 bytes  Sets: 512  Assoc: 4 way – Total size: 128K

\[2^6 \times 2^9 \times 2^2 = 2^{17} = 2^7 \times 2^{10}\]
Resolving a read request

Block size: 64 bytes  Sets: 512  Assoc: 4 way – Total size: 128K

For current AMD64 machines, only 48 of 64 bits are used.

X field of request address is ignored.
Resolving a read request

Request contents of block at 64 bit address

9 bits identify set to search (\(\log_2(512) = 9\))

Block size: 64 bytes   Sets: 512  Assoc: 4 way – Total size: 128K
Resolving a read request

Request contents of block at 64 bit address

Remaining 33 bits are matched against 33 bit tag fields

Block size: 64 bytes  Sets: 512  Assoc: 4 way – Total size: 128K
Resolving a read request

CPU

Request contents of block at 64 bit address

Remaining 33 bits are matched against 33 bit tag fields

If any of the four 33-bit tags match, then there is a cache "hit"...otherwise, the requested data is not in the cache.

Block size: 64 bytes   Sets: 512  Assoc: 4 way – Total size: 128K
Resolving a read request

Block size: 64 bytes  Sets: 512  Assoc: 4 way  – Total size: 128K
Resolving a read request

Block size: 64 bytes  Sets: 512  Assoc: 4 way – Total size: 128K

CPU

Request contents of block at 64 bit address

Remember:

Field A is not stored in the tag because we need remember only the address of the first byte cached in each line.

Field B is not stored in the tag because it’s the same for all lines in a set. Indeed, we use them to select the set.
The Memory Hierarchy
for
A Modern Intel Chip (Sandy Bridge)
The CPU has registers built in

CPU

<table>
<thead>
<tr>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
</tr>
<tr>
<td>R2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>RN</td>
</tr>
</tbody>
</table>

Main Memory
Classes of Intel-64 / AMD 64 registers

- **General purpose registers**
  - 16 registers, 64 bits each
  - Used to compute integer and pointer values
  - Used for integer call/return values to functions

- **XMM / YMM registers**
  - 16 Registers, were 128 bits each … now 256 with Sandy Bridge (YMM)
  - Used to compute float/double values, and for parallel integer computation
  - Used to pass double/float call/return values

- **X87 Floating Point registers**
  - 8 registers, 80 bits each
  - Used to compute, pass/return long double
The Intel Sandy-Bridge Memory Hierarchy

CPU

- GPR – 64 bit
- R1
- R2
- ...
- R15

- XMM / YMM / Floating 128 or 256 bit
  - XMM0
  - XMM1
  - ...
  - XMM15

- X87 Floating (old style 80 bit)
  - FP0
  - ...
  - FP7

L1 Cache – (8Kb to 128Kb)

L2 Cache – (.5 – 2 Megabytes)

L3 Cache – (2 – 8 Megabytes)

Main Memory
Modern CPUs have multi-level caches
Actual performance of multi-level cache

<table>
<thead>
<tr>
<th>coherence state</th>
<th>local</th>
<th>latency in ns on-chip</th>
<th>other socket</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1</td>
<td>L2</td>
<td>L1</td>
</tr>
<tr>
<td>modified</td>
<td>1.5</td>
<td>4.6</td>
<td>40.4</td>
</tr>
<tr>
<td>exclusive</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>forward</td>
<td></td>
<td></td>
<td>33.8</td>
</tr>
<tr>
<td>shared</td>
<td></td>
<td></td>
<td>15</td>
</tr>
</tbody>
</table>

Table 2. Latencies for accesses to various memory locations on the dual socket Intel Sandy Bridge system

From: Molka, D. et. al, Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer, MSPC ’14 Proceedings of the workshop on Memory Systems Performance and Correctness, (http://dl.acm.org/citation.cfm?doid=2618128.2618129)
Predicting Application Performance
Simple cases

- For every design we’ve studied, a program referencing memory from a contiguous area can fill each line in succession before evicting data it’s loaded
  - You can use this fact to draw approximate conclusions about programs that work on contiguous arrays
  - If the array fits in cache, then eventually it will all wind up there, and successive accesses will be fast
  - If the array is much larger than the cache and access is random, then the miss rate will be roughly proportional to \((\text{arraysize}-\text{cachesize})/\text{arraysize}\), i.e. you’ll mostly miss

- If access is sequential & not repeated (streaming), then cache size doesn’t matter! Each block will be loaded once, maybe referenced a few times, and then never used again.
  - For reasons explained later, block size might matter.
More complex cases

- In general, the only way to predict performance for a complex access pattern is to simulate it one access at a time...
  - When caches are designed, traces of actual operating system and program execution are used to test the designs

- Typically, it’s only necessary to focus on the inner loop(s) of a program, or the code that’s accessing most data

- Often, this code will involve combinations of simple access patterns about which you can make educated guesses
  - Some pattern of access due to instruction fetches
  - Maybe a combination of predictable data structure accesses that, if you’re lucky, all fit in the cache together,
  - Etc.
Having said all that...

- **You should almost never design your program to work with a particular cache mapping or associativity**
- For maximum performance, just keep data structures small, and localize access to the extent practical
- Be alert to performance anomalies that may trace to details of cache design
  - Effects relating to cache size are common (your array fits or it doesn’t)
  - Effects relating to mapping & set associativity are rare and can usually be ignored, but very rarely they will cause dramatic effects that you should recognize
  - If you’re building a performance critical application, test it on a range of hardware representative of actual deployment, and learn to use tools that can report on cache performance
How C Language Data is Represented
Why align to “natural sizes”?

- Memory systems delivers aligned to cache blocks
- CPU loads aligned words or blocks from cache
- Data that spans boundaries needs two accesses!
- Naturally aligned data never unnecessarily crosses a boundary
- Intel/AMD alignment rules
  - Pre AMD 64: unaligned data allowed but slower
  - AMD 64: data must be aligned as shown on previous slides

http://www.cs.tufts.edu/comp/40/readings/amd64-abi.pdf#page=13
## Sizes and alignments

<table>
<thead>
<tr>
<th>C Type</th>
<th>Width</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long, pointer, long long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Do you see a pattern?

## Sizes and alignments

<table>
<thead>
<tr>
<th>C Type</th>
<th>Width</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long, pointer, long long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>10</td>
<td>16</td>
</tr>
</tbody>
</table>

*But what about this?*

Alignment of C Language Data

- **Structures**
  - *Wasted space used to “pad” between struct members if needed to maintain alignment*
  - Therefore: when practical, put big members of larger primitive types ahead of smaller ones!
  - Structures sized for worst-case member
  - *Arrays of structures will have properly aligned members*

- **malloc return values always multiple of 16**
  - `structp = (struct s *)malloc(sizeof(struct s))` is always OK

- **Argument area on stack is 16 byte aligned**
  - `%rsp-8` is multiple of 16 on entry to functions
  - Extra space “wasted” in caller frame as needed
  - *Therefore: called function knows how to maintain alignment*
Putting it all together: arrays of structs

Consider this code fragment and assume a 64 byte cache block:

```c
struct s {double d, e; int x; float f, g;};
struct s arr[100000];  // array of structs

for (i = 0; i < 100000; i++) {...access arr[i].x...}
```

- Note that, due to padding, sizeof(struct s) == 32. (Where are the padding bytes?)
- Although the loop is referencing only 4 bytes when accessing arr[i].x, an entire 64 byte cache block is loaded on each miss
- Thus, the miss rate for this loop is likely 50%. First a line is loaded due to a miss, but that load brings in two copies of the 32 byte structure, so the next access is a hit
- Note that the successive accesses have a stride of 32 bytes, which is the size of the struct