2019/2020, 1st quarter

INFOMOV: Optimization & Vectorization / *OptmzdSummary*

#2 – Caching

Author: Jacco Bikker

TL;DR

A modern processor executes billions of instructions per second. Those instructions need data, and in fact, they *are* data. Getting all that data to the CPU efficiently is a tough challenge, that requires a complex memory system consisting of RAM and caches. Understanding this system allows us to significantly improve the performance of applications that are memory bound.

In this document we first explore the basics of caching. After this we investigate how software can be adapted to better match the characteristics of the memory system.

The Problem with Memory /ˈmem(ə)ri/

A processor that runs at 4Ghz (e.g., Intel's i7-6700k or AMD's FX-8350) potentially needs data every quarter of a nanosecond. An interesting thought experiment is thus: if the processor is to receive instructions and data from RAM at that rate, how far can the memory physically be from the processor?

The answer is shocking: if light travels $c = 299{,}792{,}458$ meters per second ($9.461 \cdot 10^{12}$ kilometers per year), it will travel almost 7.5 *centimeters* in a quarter of a nanosecond. So, a processor that asks for data and memory that delivers the data cannot be further apart than 3.75cm, under ideal circumstances. Note that the speed of light in an electronic circuit is lower (about $0.95c$).

There are several ways to deal with this:

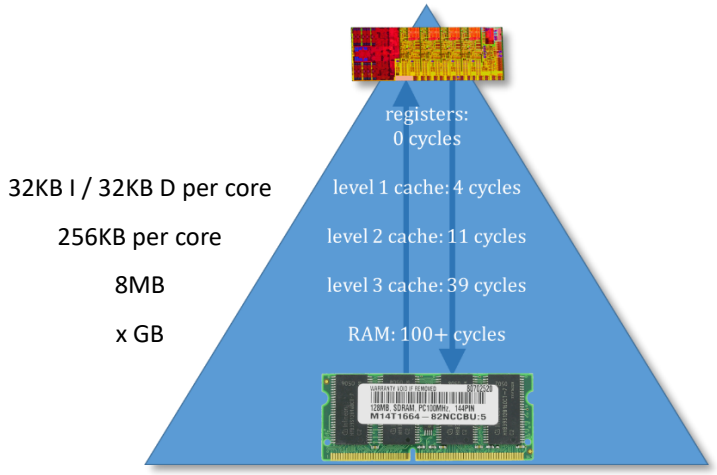
1. we could allow data to travel much longer than a single cycle;
2. we can store a small amount of memory *on* the CPU.

Regarding option 1: according to AIDA64, the 'Dual Channel DDR4-2133 SDRAM' in my desktop needs 67.6ns to get data to the 3.5Ghz CPU. That is a latency of *hundreds* of cycles. Once the stream flows, things get better: the *bandwidth* of the SDRAM is 27.8GB/s. That is however only useful if we operate on a stream. For random access, the latencies hit us at full force.

Caches

To bridge the performance gap between RAM and CPU we use a *memory hierarchy*. The idea is to keep some data on the CPU in a small amount of memory consisting of registers and a *cache:* a small amount of high-speed memory that stores a subset of the data in RAM. When the CPU requests data from RAM, this request will go via the cache. If the data is in the cache the CPU receives it immediately. Otherwise, the data is brought in from RAM and stored in the cache. The CPU suffers the full latency, but at least next time the same data will be available more quickly.

A modern CPU uses several levels of cache. The level-1 cache is the fastest, but also the smallest. Data that is not stored in level-1 cache may be found in the larger level-2 cache, at a small performance penalty. After that, there is the level-3 cache, which is even larger, but also even slower. The memory is now the last resort.



Intel i7-9xx memory hierarchy (quad core)

There are some conditions for this to work. First-time memory access is as slow as ever; only *reuse* is faster. And: caches are relatively small. If we read a lot of data, chances are that we need to *evict* data that we still want to use.

To better understand how we can optimize software for the memory hierarchy we need to understand how caches work.

Fully Associative

The most basic caching scheme is the *fully associative cache*. It's conveniently described as a C data structure:

```
1  struct CacheLine
2  {
3     uint address; // 32-bit for max. 4GB
4     BYTE data;
5     bool valid;
6  };
7  CacheLine cache[256]; // 256 bytes
```

This cache can hold 256 bytes, in 256 lines of one byte. Whenever the CPU wants to access a byte, we check each of the 256 lines to see if the requested address is represented in the cache. If it isn't present, it gets fetched from the next level of the memory hierarchy.

Note the presence of a `valid` field. A line only becomes valid once we read data in it. On the other hand, an invalid line is a line we can safely overwrite. Once we run out of these, we need to remove a line. Which line we evict depends on the *eviction policy*.

A slightly better cache is shown below:

```
1  struct CacheLine
2  {
3     uint tag; // 30-bit for max. 4GB
4     uint data;
5     bool valid, dirty;
6  };
7  CacheLine cache[64]; // 256 bytes
```

The changes are in bold. First of all, a cacheline now stores four *bytes* rather than one. This is because the CPU rarely accesses a single byte: integers, floats and 32-bit pointers are all 4-bytes. Reading an integer can now be done in a single transaction.

The second change is the address storage, which is now labeled **`tag`**. Since each line now stores 4 bytes, every address must be a multiple of 4. In binary, that means that the lowest two bits are always zero. We don't need to store them; just the top 30 bits suffice.

The CPU may still want to access individual bytes. For this, the requested address is split in a 30-bit tag and a 2-bit offset. The offset is used to select a byte in the 4-byte cacheline.

The final change is the **`dirty`** flag. When data is evicted, it must be written back to the next level, but only if it was changed. Otherwise, we can just discard it, which is much faster. The `dirty` field is thus set to `false` when we fill a line from the next level; we set it to `true` whenever the CPU modifies the cached value.

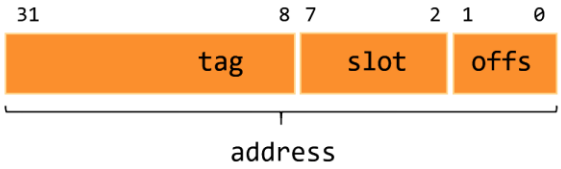
Direct Mapped Cache

The fully associative cache has one major problem: it doesn't scale. For the tiny 256-byte cache we already needed to compare 64 addresses. Remember that this must be implemented in hardware, and that the latency must be as low as possible.

We can solve this using a direct mapped cache.

In a direct mapped cache, the index of the cacheline that will be used for a particular address is calculated from the address. Each address can thus be stored (or found) in only one cacheline, and we no longer have to search.

Our cache so far used 64 lines, which can be addressed using 6 bits. We extract these directly from the address:



Note that this reduces the *tag* to only 24 bits.

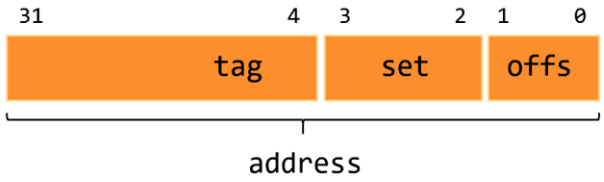
The direct mapped cache introduces a new problem: *collisions.* Two addresses that have the same bits $2..7$ will map to the same cacheline. This is the case for addresses that are 256 bytes apart. Imagine a 256×800 2D array of bytes: processing a column of this array will overwrite a single cacheline each time, while all other cachelines remain unused.

N-Way Set Associative Cache

We can combine the concepts of a fully associative cache and a direct mapped cache to obtain the N-way set associative cache, which is commonly used in modern processors. In the N-way set associative cache, we group cachelines in sets:

```
1  struct CacheLine
2  {
3     uint tag; // 30-bit for max. 4GB
4     uint data;
5     bool valid, dirty;
6  };
7  CacheLine cache[16][4]; // 256 bytes in 16 sets of 4 lines of 4 bytes
```

The set is again obtained directly from the address:

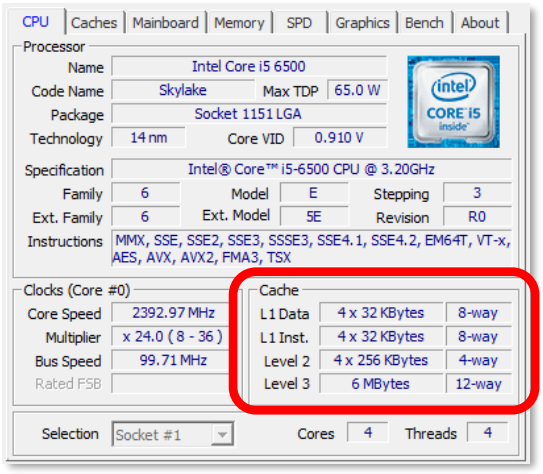


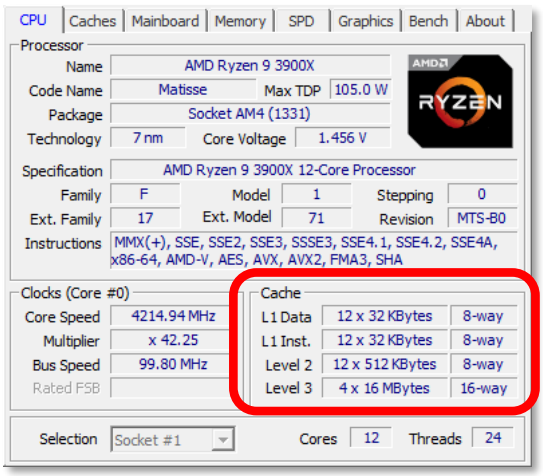
This time, determining the set is not enough: any of the four cachelines in the set may contain the data we are looking for. For $N = 4$, this can be done efficiently in hardware.

Actual Hardware

It's time to replace the numbers in the previous sections with numbers used in actual hardware.

As mentioned, the *N*-way set associative cache is common: both Intel and AMD use this scheme for their L1, L2 and L3 caches. According to CPU-Z, Intel's i5 6500 uses a 32KB 8-way set associative cache for instructions and data at the first level. The L2 cache is 4-way, and the large L3 cache is 12-way. AMD's Ryzen 3000 uses 8-way for L1 and L2, and 16-way for L3.

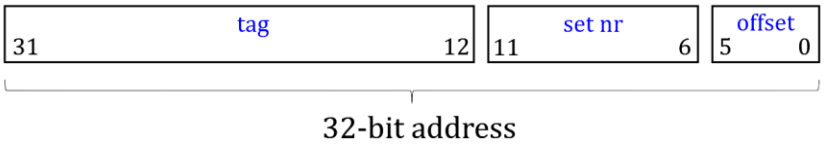




On modern CPUs, a single cache line is 64 bytes. Fetching a single integer thus triggers the transfer of 15 additional integers. This works well with sequential memory access, where a single cache miss is followed by several hits.

With these numbers in place, we can make the L1 data cache a bit more concrete:

32KB, 8-way, 64 bytes per line means that we have sets of 8*64=512 bytes each. The cache thus consists of 64 sets. Memory addresses are broken down as follows:



We can now also see that addresses that are 4096 bytes apart map to the same set. Since we have 8 lines in each set, this will not immediately lead to collisions, but certain access patterns (e.g., column-wise processing of a 1024 wide 2D table) will still lead to under-utilization of the cache.

Optimizing for Caches

To optimize for caches, we must optimize our data. Specifically:

Data size Cache space is finite, especially in the fast L1 cache. Reducing data size increases the chance of cache hits in the top of the memory hierarchy.

Access patterns If the data is not small at all, we can still exploit the caches by working on a localized set which does fit in the cache.

Cacheline size A cache line is 64 bytes. This means that reading the first element of a 64-byte struct gets all the other fields in L1 cache as well. If the struct is only 68 bytes, it requires two cacheline transfers. Shaving off four bytes can have a significant impact.

Cacheline alignment Each cacheline holds data for a memory address that is a multiple of 64. If a 64-byte struct does not start on a cacheline boundary, will require 2 transfers. We need to carefully align our data in memory, even if that means adding 4 bytes to a 60-byte struct.

Sharing Intel and AMD CPUs use a L1 cache per core. This means that it becomes possible for data to reside in *two* L1 caches. Changing one of them has consequences for the other.

The End

This material is part of the Optimization & Vectorization course of the MGT master program at the Utrecht University in the Netherlands. More information at:

http://www.cs.uu.nl/docs/vakken/mov

Utrecht University