Welcome!
Today's Agenda:

- The Problem with Memory
- Cache Architectures
- Consequences
- A Handy Guide *(to pleasing the cache)*
Introduction

Feeding the Beast

Let's assume our CPU runs at 4Ghz. What is the maximum physical distance between memory and CPU if we want to retrieve data every cycle?

Speed of light (vacuum): 299,792,458 m/s
Per cycle: ~0.075 m → ~3.75cm back and forth.

In other words: we cannot physically query RAM fast enough to keep a CPU running at full speed.
Introduction

Feeding the Beast

Sadly, we can’t just divide by the physical distance between CPU and RAM to get the cycles required to query memory.

Factors include (stats for DDR4-3200/PC4-25600):

- **RAM runs at a much lower clock speed than the CPU**
  - 25600 here means: theoretical bandwidth in MB/s
  - 3200 is the number of transfers per second (1 transfer=64bit)
  - We get two transfers per cycle, so actual I/O clock speed is 1600Mhz
  - DRAM cell array clock is \(~1/4\)th of that: 400Mhz.

- **Latency between query and response:** 20-24 cycles.
Introduction

Feeding the Beast

Sadly, we can’t just divide by the physical distance between CPU and RAM to get the cycles required to query memory.

Factors include (stats for DDR4-3200/PC4-25600):

- **Latency between query and response:** 20-24 cycles.

**SRAM:**

- Maintains data as long as $V_{dd}$ is powered (no refresh).
- Bit available on $BL$ and $BL$ as soon as $WL$ is raised (fast).
- Six transistors per bit ($$).
- Continuous power ($$$).
Introduction

Feeding the Beast

Sadly, we can’t just divide by the physical distance between CPU and RAM to get the cycles required to query memory.

Factors include (stats for DDR4-3200/PC4-25600):


**DRAM:**
- Stores state in capacitor C.
- Reading: raise AL, see if there is current flowing.
- Needs rewrite.
- Draining takes time.
- Slower but cheap.
- Needs refresh.
Introduction

Feeding the Beast

Sadly, we can't just divide by the physical distance between CPU and RAM to get the cycles required to query memory.

Factors include (stats for DDR4-3200/PC4-25600):

Introduction

Feeding the Beast

Sadly, we can't just divide by the physical distance between CPU and RAM to get the cycles required to query memory.

Additional delays may occur when:

- Other devices than the CPU access RAM;
- DRAM must be refreshed every 64ms due to leakage.

For a processor running at 2.66GHz, latency is roughly 110-140 CPU cycles.

Introduction

Feeding the Beast

“We cannot physically query RAM fast enough to keep a CPU running at full speed.”

How do we overcome this?

We keep a copy of frequently used data in fast memory, close to the CPU: the cache.
Introduction

The Memory Hierarchy – Core i7-9xx (4 cores)

- **Registers**: 0 cycles
- **Level 1 Cache**: 4 cycles
- **Level 2 Cache**: 11 cycles
- **Level 3 Cache**: 39 cycles
- **RAM**: 100+ cycles

- **32KB I / 32KB D per core**
- **256KB per core**
- **8MB**
- **x GB**

**Infomov – Lecture 8 – “Caching (1)”**
Introduction

Caches and Optimization

Considering the cost of RAM vs L1$ access, it is clear that the cache is an important factor in code optimization:

- Fast code communicates mostly with the caches
- We still need to get data into the caches
- But ideally, only once.

Therefore:

- The working set must be small;
- Or we must maximize data locality.
Today’s Agenda:

- The Problem with Memory
- Cache Architectures
- Consequences
- A Handy Guide *to pleasing the cache*
Cache Architecture

The simplest caching scheme is the fully associative cache.

```c
struct CacheLine {
    uint address; // 32-bit for 4G
    uchar data;
    bool valid;
};

CacheLine cache[256];
```

This cache holds 256 bytes.

Notes on this layout:

- We will rarely need 1 byte at a time
- So, we switch to 32-bit values
- We will rarely read those at odd addresses
- So, we drop 2 bits from the address field.
The simplest caching scheme is the **fully associative cache**.

```c
struct CacheLine {
    uint tag;       // 30 bit for 4G
    uint data;
    bool valid, dirty;
};
CacheLine cache[64];
```

This cache holds 64 dwords (256 bytes).
Architectures

Cache Architecture

The simplest caching scheme is the **fully associative cache**.

```c
struct CacheLine {
    uint tag;     // 30 bit for 4G
    uint data;
    bool valid, dirty;
};
CacheLine cache[64];
```

This cache holds 64 dwords (256 bytes).

Single-byte read operation:

```c
for ( int i = 0; i < 64; i++ )
    if (cache[i].valid)
        if (cache[i].tag == tag)
            return cache[i].data[offs];
uint d = RAM[tag].data; // cache miss
WriteToCache( tag, d );
return d[offs];
```
Architectures

Cache Architecture

The simplest caching scheme is the **fully associative cache**.

```c
struct CacheLine
{
    uint tag;       // 30 bit for 4G
    uint data;
    bool valid, dirty;
};
CacheLine cache[64];
```

This cache holds 64 dwords (256 bytes).

One problem remains... We store one byte, but the slot stores 4. What should we do with the other 3?

Single-byte write operation:

```c
for ( int i = 0; i < 64; i++ )
    if (cache[i].valid)
        if (cache[i].tag == a)
            cache[i].data[offs] = d;
            cache[i].dirty = true;
            return;

    if ( !cache[i].valid)
        cache[i].tag = a;
        cache[i].data[offs] = d;
        cache[i].valid|dirty = true;
        return;

i = BestSlotToOverwrite();
if (cache[i].dirty) SaveToRam(i);
    cache[i].tag = a;
    cache[i].data[offs] = d;
    cache[i].valid|dirty = true;
```
The best slot to overwrite is the one that will not be needed for the longest amount of time. This is known as Bélády’s algorithm, or the *clairvoyant* algorithm.

Alternatively, we can use:

- LRU: least recently used
- MRU: most recently used
- Random Replacement
- LFU: Least frequently used
- ...

AMD and Intel use ‘pseudo-LRU’ (until Ivy Bridge; after that, things got complex*).

---

*In case this isn’t obvious: this is a hypothetical algorithm; the best option if we actually had a crystal orb.*

---

[http://blog.stuffedcow.net/2013/01/ivb-cache-replacement](http://blog.stuffedcow.net/2013/01/ivb-cache-replacement)
The Problem with Being Fully Associative

Read / Write using a fully associative cache is $O(N)$: we need to scan each entry. This is not practical for anything beyond 16~32 entries.

An alternative scheme is the direct mapped cache.
Direct Mapped Cache

```c
struct CacheLine {
    uint tag;       // 24 bit for 4G
    uint data;
    bool dirty, valid;
};
```

CacheLine cache[64];

This cache again holds 256 bytes.

In a direct mapped cache, each address can only be stored in a single cache line.

Read/write access is therefore \( O(1) \).

For a cache consisting of 64 cache lines:

- Bit 0 and 1 still determine the offset within a slot;
- 6 bits are used to determine which slot to use;
- The remaining 24 bits form the tag.
Direct Mapped Cache

In general:

- Bits 0..N-1 are used as offset in a cache line;
- Bits N..M-1 are used as slot index;
- Bits M..31 are used as tag.

\[
N = \log_2(\text{cache line width})
\]
\[
M = \log_2(\text{number of slots in the cache})
\]
The Problem with Direct Mapping

In this type of cache, each address maps to a single cache line, leading to O(1) access time. On the other hand, a single cache line ‘represents’ multiple memory addresses.

This leads to a number of issues:

- A program may use two variables that occupy the same cache line, resulting in frequent cache misses (collisions);
- A program may heavily use one part of the cache, and underutilize another.
Architectures

N-Way Set Associative Cache

```c
struct CacheLine {
    uint tag;
    uint data;
    bool valid, dirty;
};
CacheLine cache[16][4];
```

This cache again holds 256 bytes.

In an N-way set associative cache, we use N slots (cache lines) per set.
N-Way Set Associative Cache

```c
struct CacheLine {
    uint tag; // 28 bit for 4G
    uint data;
    bool valid, dirty;
};

CacheLine cache[16][4];
```

This cache again holds 256 bytes.

In an N-way set associative cache, we use N slots (cache lines) per set.

When reading / writing data, we check each of the N slots that may contain the data.

Example: Address 0x00FF1004

Offset: lowest 2 bits \(\equiv 0\).
Set: next 2 bits \(\equiv 1\).
Tag: remaining bits.
Caching Architectures

The Intel i7 processors use three on-die caches:
L1: 32KB 4-way set associative instruction cache + 32KB 8-way data cache per core
L2: 256KB 8-way set associative cache per core
L3: 2MB x cores global 16-way set associative cache.

The AMD Phenom also uses three on-die caches:
L1: 64KB 2-way set associative (32+32) per core
L2: 512KB 16-way set associative per core
L3: 1MB x cores global 48-way set associative cache.

Both AMD and Intel currently use 64 byte cache lines.
Architectures

32KB, 4-Way Set Associative Cache

```c
struct CacheLine {
    uint tag;  // 19 bit for 4G
    uchar data[64];
    bool valid, dirty;
};
CacheLine cache[128][4];
```

This cache holds 32768 bytes in 512 cachelines, organized in 128 sets of 4 cachelines.
Today's Agenda:

- The Problem with Memory
- Cache Architectures
- Consequences
- A Handy Guide *(to pleasing the cache)*
Consequences

64 bytes per cache line

Theoretical consequence:

- If address $X$ is pulled into the cache, so is $(X+1, ..., X+63)$.

Example*:

```java
int arr = new int[64 * 1024 * 1024];
// loop 1
for (int i = 0; i < 64 * 1024 * 1024; i++) arr[i] *= 3;
// loop 2
for (int i = 0; i < 64 * 1024 * 1024; i += 16) arr[i] *= 3;
```

Which one takes longer to execute?

Consequences

64 bytes per cache line

Theoretical consequence:

- If the object you’re querying straddles the cache line boundary, you may suffer not one but two cache misses.

Example:

```c
struct Pixel { float r, g, b; }; // 12 bytes
Pixel screen[768][1024];
```

Assuming pixel (0,0) is aligned to a cache line boundary, the offsets in memory of pixels (0,1..5) are 12, 24, 36, 48, 60, ... . Walking column 5 will be very expensive.
Consequences

"Access Patterns Matter"
Consequences

“Access Patterns Matter”
Consequences

“Access Patterns Matter”

Improving data locality: z-order / Morton curve

Method:

\[
\begin{align*}
X &= 1 1 0 0 0 0 1 0 1 1 0 1 1 0 1 1 0 1 \\
Y &= 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0
\end{align*}
\]

\[
\text{address} = 1101101000111001110011111001
\]
Today’s Agenda:

- The Problem with Memory
- Cache Architectures
- Consequences
- A Handy Guide *(to pleasing the cache)*
Easy Steps

How to Please the Cache

Or: “how to evade RAM”

1. Keep your data in registers

Use fewer variables
Limit the scope of your variables
Pack multiple values in a single variable
Use floats and ints (they use different registers)
Compile for 64-bit (more registers)
Arrays will never go in registers

Unions will never go in registers
Easy Steps

How to Please the Cache

Or: “how to evade RAM”

2. Keep your data local

Read sequentially
Keep data small
Use tiling / Morton order
Fetch data once, work until done (streaming)
Reuse memory locations
How to Please the Cache

Or: “how to evade RAM”

3. Respect cache line boundaries

Use padding if needed

Don’t pad for sequential access

Use aligned malloc / __declspec align

Assume 64-byte cache lines
Easy Steps

How to Please the Cache

Or: “how to evade RAM”

4. Advanced tricks

Prefetch
Use a prefetch thread (theoretical...)
Use *streaming writes*
Separate mutable / immutable data
Easy Steps

How to Please the Cache

Or: “how to evade RAM”

5. Be informed

Use the profiler!
Today’s Agenda:

- The Problem with Memory
- Cache Architectures
- Consequences
- A Handy Guide (*to pleasing the cache*)
END of “Caching (1)”

next lecture: “GPGPU (3)”