/INFOMOV/

Optimization & Vectorization

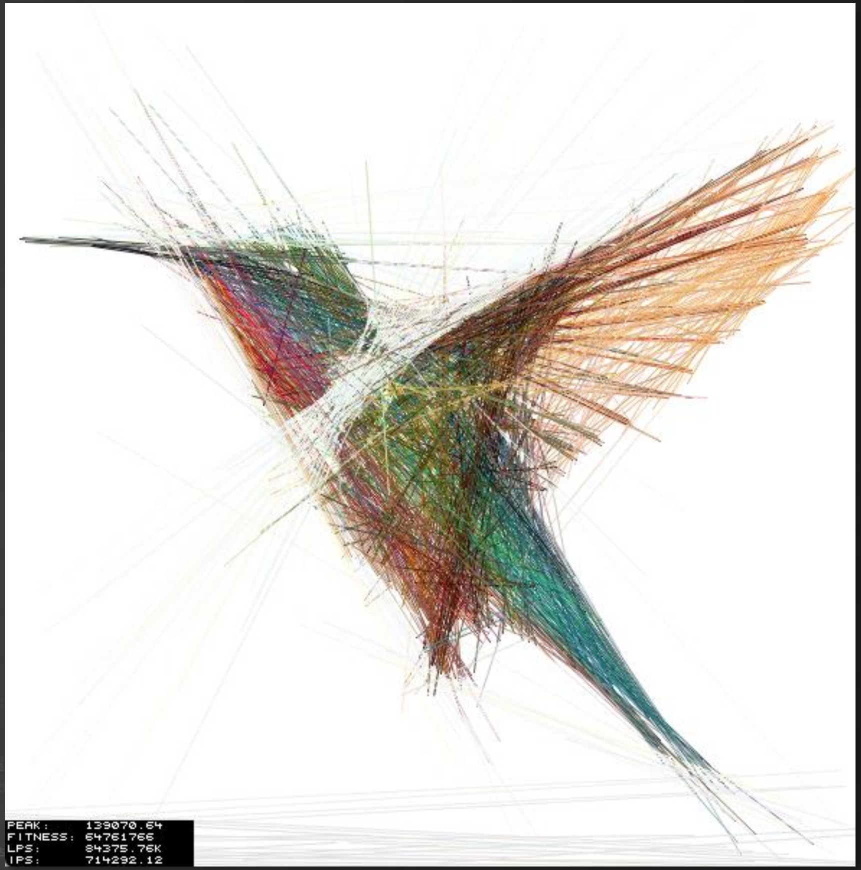
J. Bikker - April – June 2024 - Lecture 4: “Caching (1)”

Welcome!

Today’s Agenda:

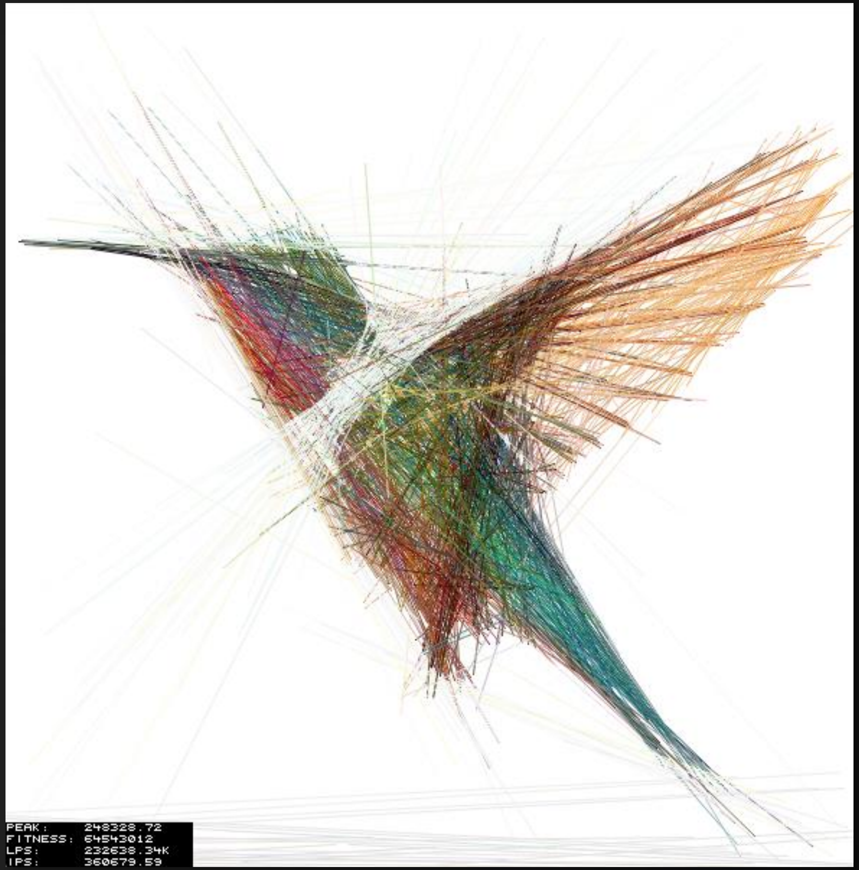
- The Problem with Memory
- Cache Architectures

```
rics
& (depth < MAXDEPTH))

t = inside ? 1 : 1.2f
nt = nt / nc, ddn
os2t = 1.0f - nnt * nnt
D, N );
0)

at a = nt - nc, b = nt + nc
at Tr = 1 - (R0 + (1 - R0)
Tr) R = (D * nnt - N * (ddn

E * diffuse;
= true;

-
efl + refr)) && (depth < MAXDEPTH))

D, N );
refl * E * diffuse;
= true;


MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely
df;
radiance = SampleLight( &rand, I, &L, &li
e.x + radiance.y + radiance.z) > 0) && (d

w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Ps
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance

andom walk - done properly, closely following Smal
vive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
urvive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```



```
PEAK:     139070.64
FITNESS:  64761766
LPS:      84375.76K
IPS:      714292.12
```

139070



```
PEAK:     248328.72
FITNESS:  64543012
LPS:      232638.34K
IPS:      360679.59
```

248328

× 1.7856

Today’s Agenda:

- The Problem with Memory
- Cache Architectures

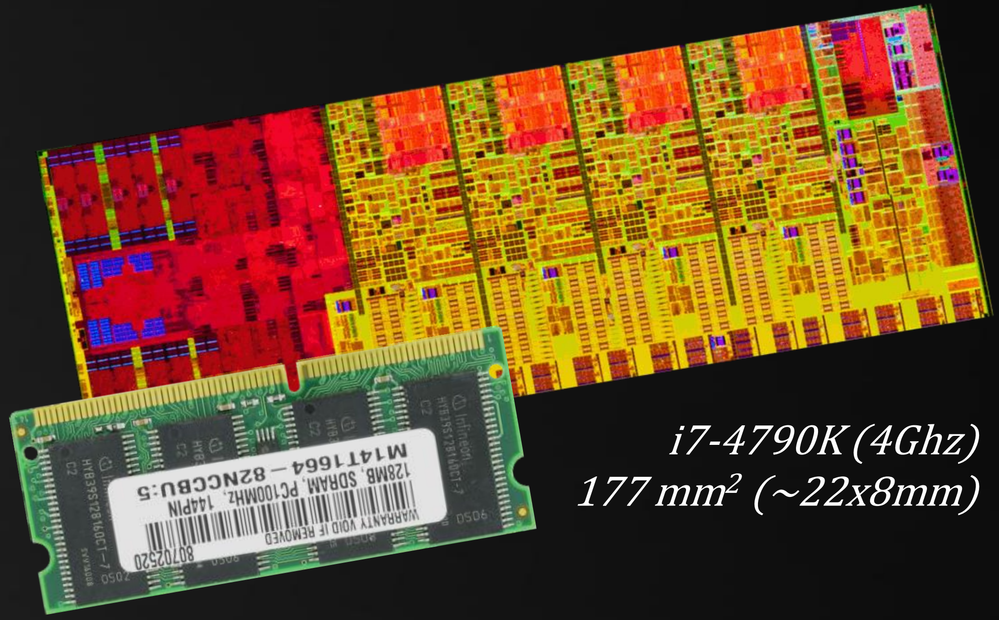
Introduction

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.



i7-4790K (4Ghz)
177 mm^2 (~22x8mm)

Note: signals in electronics do *not* propagate at the speed of light:
https://en.wikipedia.org/wiki/Velocity_factor#Typical_velocity_factors
TL;DR: It’s between 0.6c .. 0.99c.

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 ~1/4th of that: 400Mhz.
- Latency between query and response: 20-24 cycles.

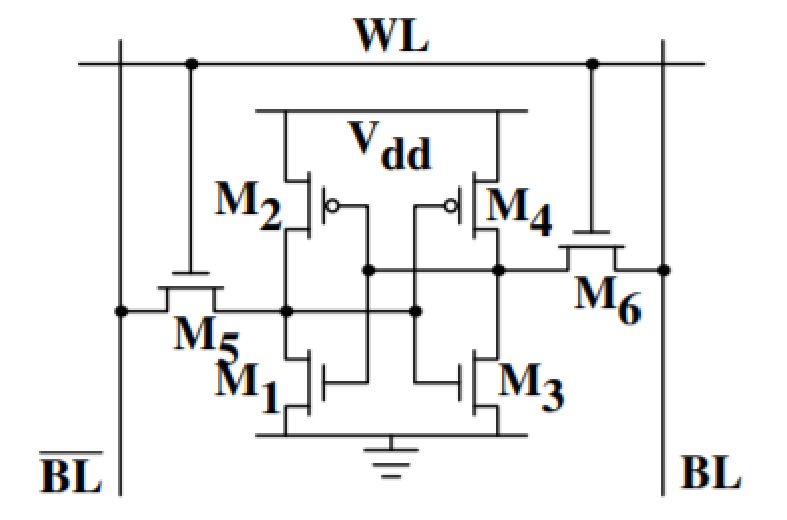
Introduction

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 ($$$).

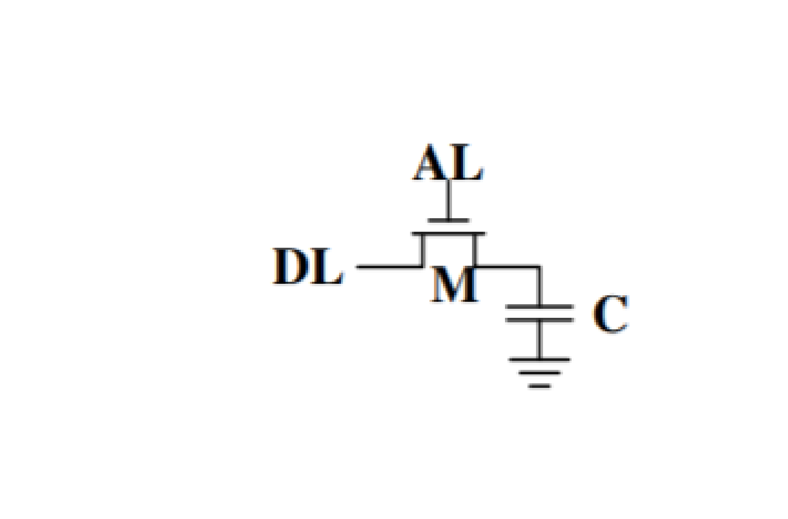
Introduction

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):

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



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.

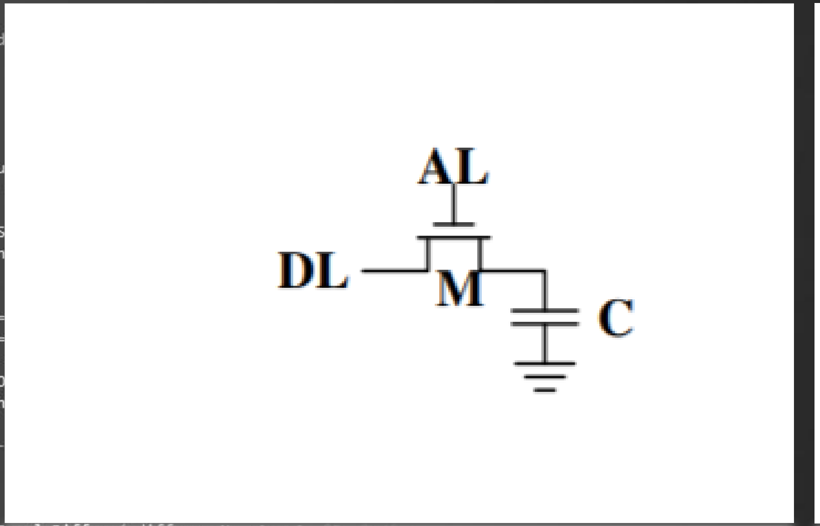
Introduction

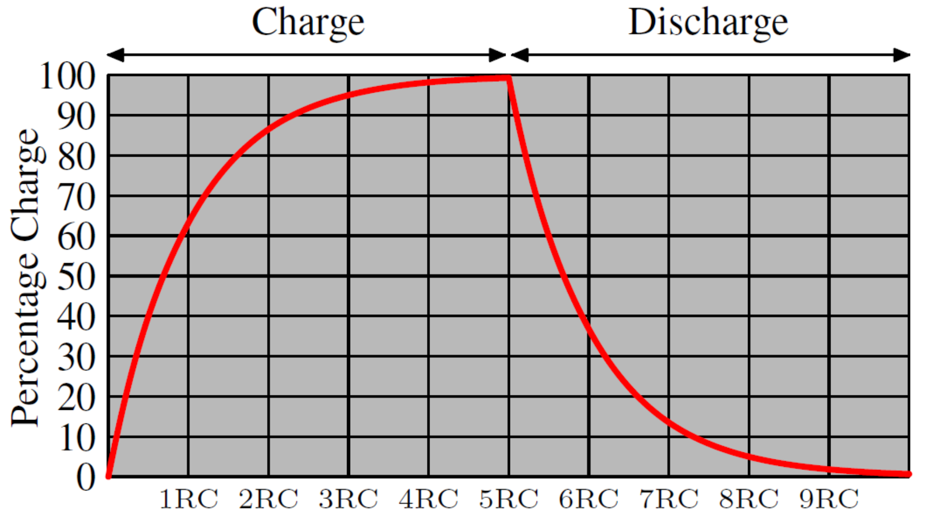
Feeding the Beast

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):

- 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.

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.

Details in: “What Every Programmer Should Know About Memory”, Chapter 2.

Introduction

Feeding the Beast

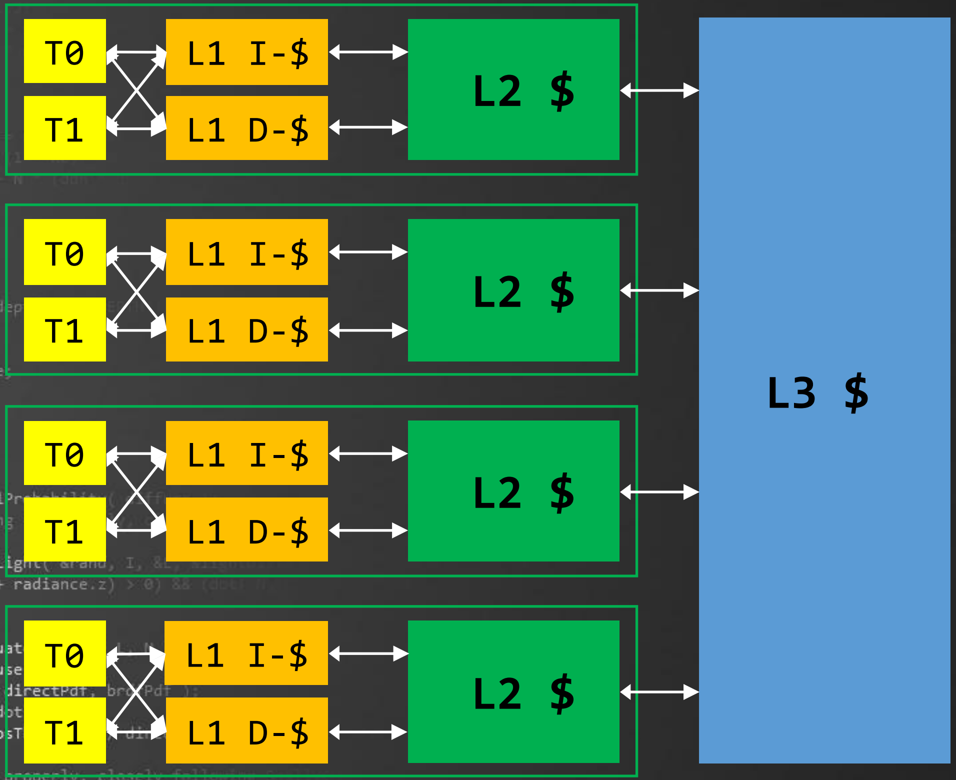
“We cannot physically query RAM fast enough to keep a CPU running at full speed.”

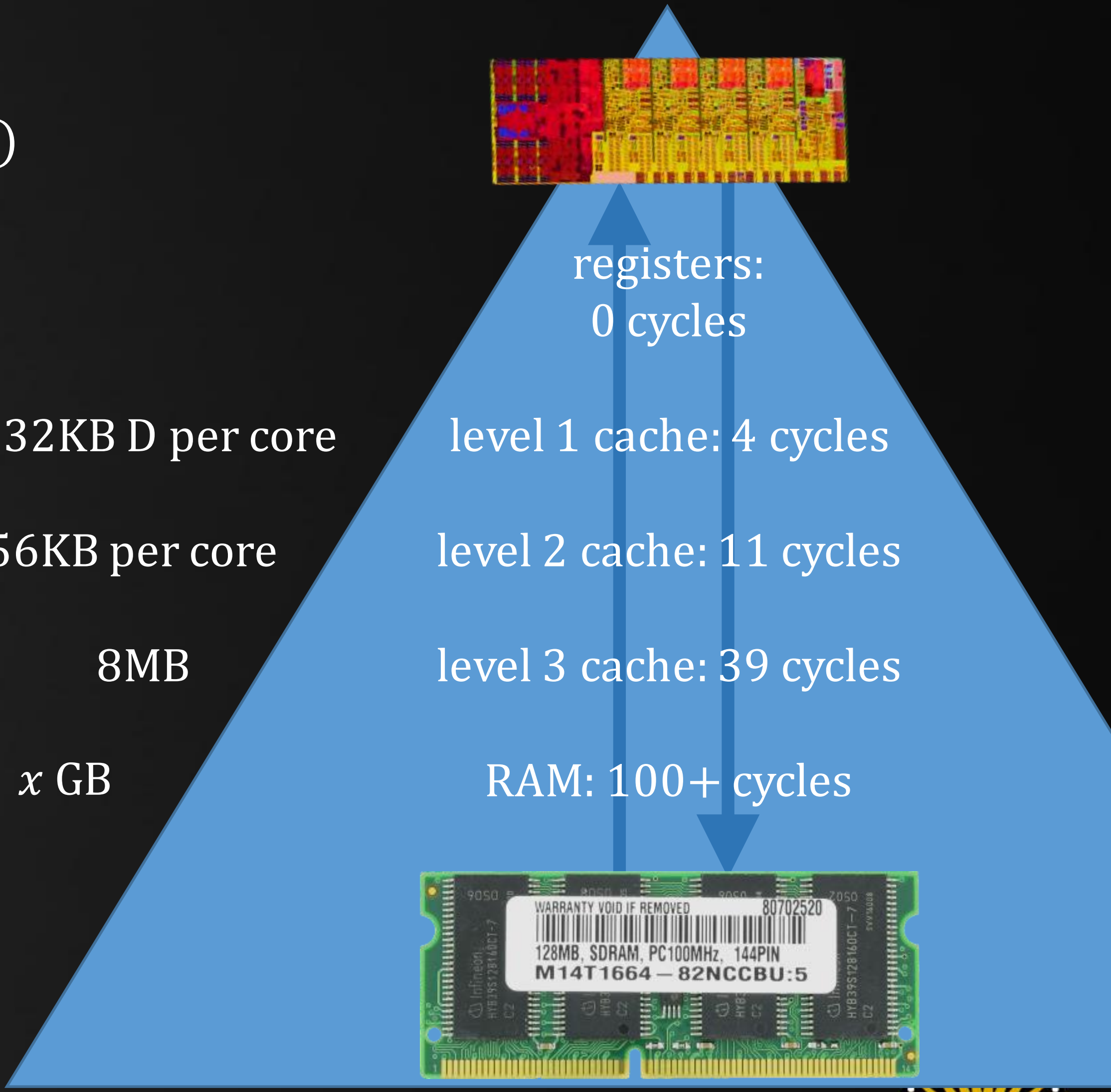
How do we overcome this?

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

Introduction

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





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

Architectures

Cache Architecture

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

```
struct CacheLine
{
    uint address; // 32-bit for 4G
    uchar data;
    bool valid;
};
CacheLine cache[256];
```

This cache holds 256 bytes.

This cache has an administrative overhead of 33x256 bits = ~8KB.

address	data	valid
0x00000000	0xFF	0
0x00000000	0xFF	0
0x00000000	0xFF	0
0x00000000	0xFF	0
0x00000000	0xFF	0
…	…	…
0x00000000	0xFF	0

Notes on this layout:

- We will rarely need 1 byte at a time
- So, we switch to 32bit values
- We will never read those at addresses that are not a multiple of 4
- So, we drop 2 bits from the address field (and: we rename it to ***tag***).

Architectures

Cache Architecture

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

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

This cache holds 64 dwords (256 bytes).

This time, the administrative overhead is 32 * 64 = 2KB.

tag	data	valid	dirty
0x00000000	0xFFFFFFFF	0	0
0x00000000	0xFFFFFFFF	0	0
0x00000000	0xFFFFFFFF	0	0
0x00000000	0xFFFFFFFF	0	0
0x00000000	0xFFFFFFFF	0	0
…	…		
0x00000000	0xFFFFFFFF	0	0

Architectures

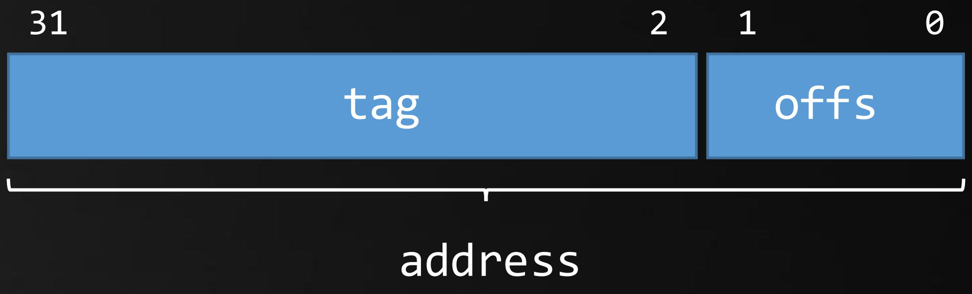
Cache Architecture

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

```
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:



```
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*.

```
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 write operation:

```
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;

for ( int i = 0; i < 64; i++ )
   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;
```

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

Architectures

BestSlotToOverwrite() ?

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.

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

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*).

*: http://blog.stuffedcow.net/2013/01/ivb-cache-replacement

Architectures

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*.

Architectures

Direct Mapped Cache

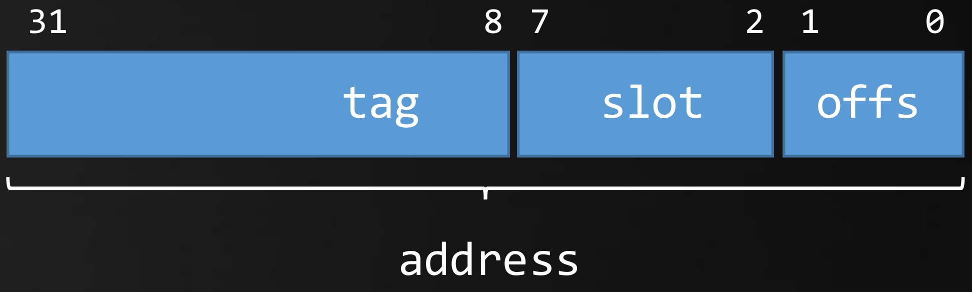
```
struct CacheLine
{
    uint tag;     // 24 bit for 4G
    uint data;
    bool dirty, valid;
};
CacheLine cache[64];
```

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).

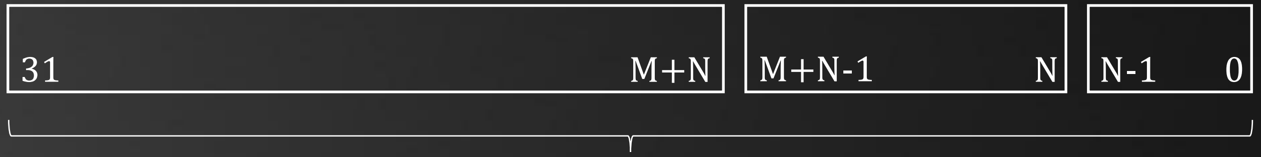
For a cache consisting of 64 cache lines:



- 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.

Architectures

Direct Mapped Cache



32-bit address

In general:

$$N = \log_2(cache\ line\ width)$$
$$M = \log_2(number\ of\ slots\ in\ the\ cache)$$

- 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.

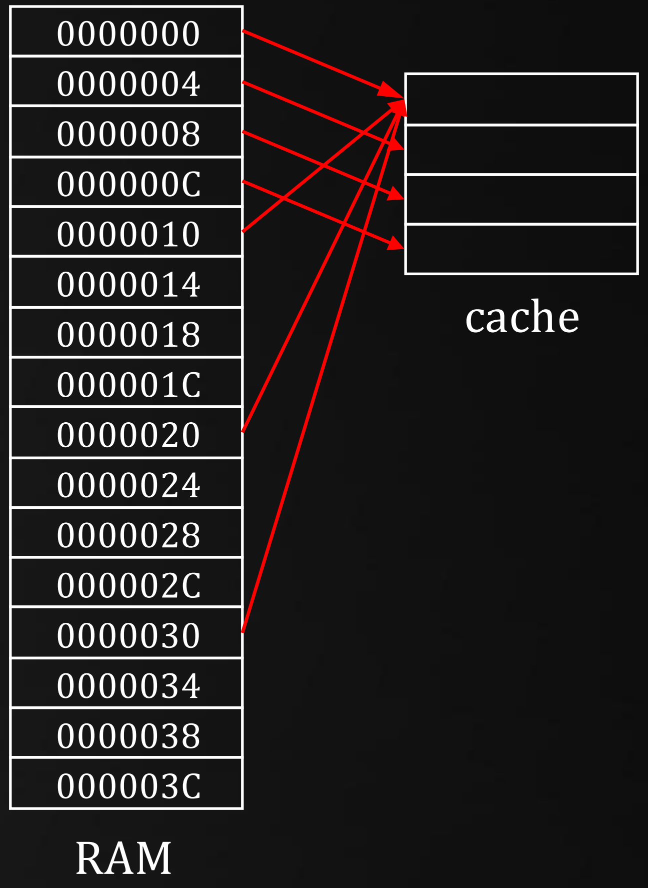
Architectures

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.

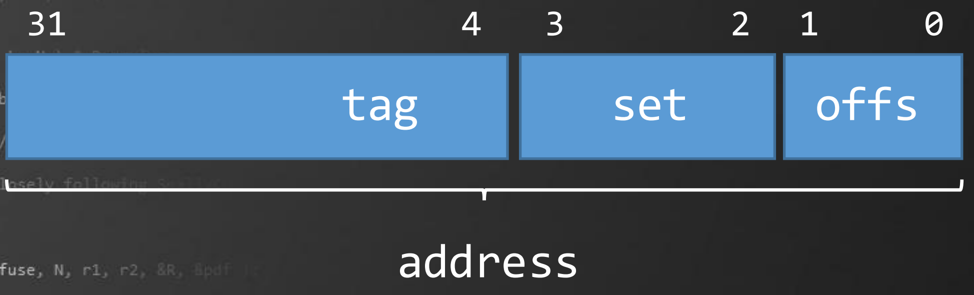


Architectures

N-Way Set Associative Cache

```
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.

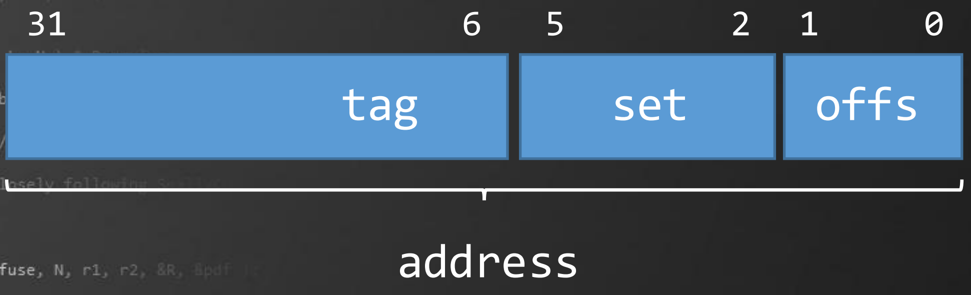
set	slot 0	slot 1	slot 2	slot 3
0000				
0001				
0002				
0003				
0004				
0005				
0006				
0007				
0008				
0009				
0010				
0011				
0012				
0013				
0014				
0015				

Architectures

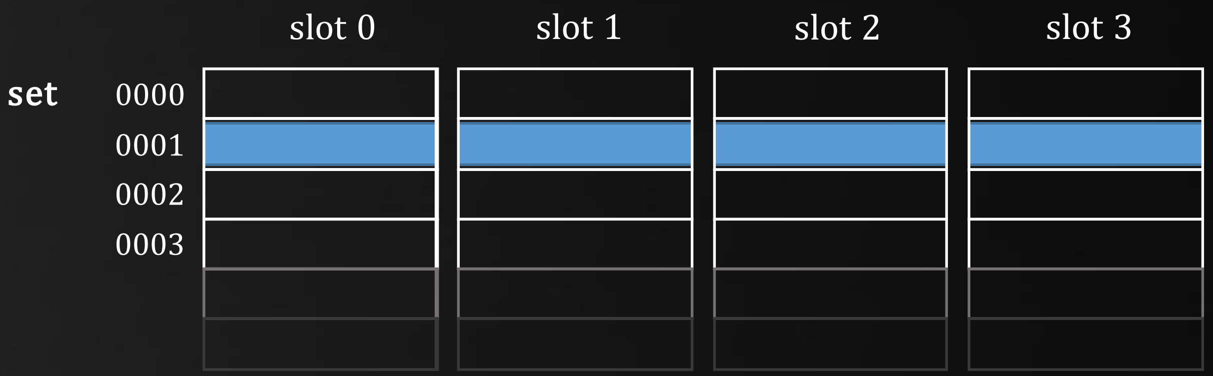
N-Way Set Associative Cache

```
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 → 0.
Set: next 4 bits → 1.
Tag: remaining bits.

Architectures

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.

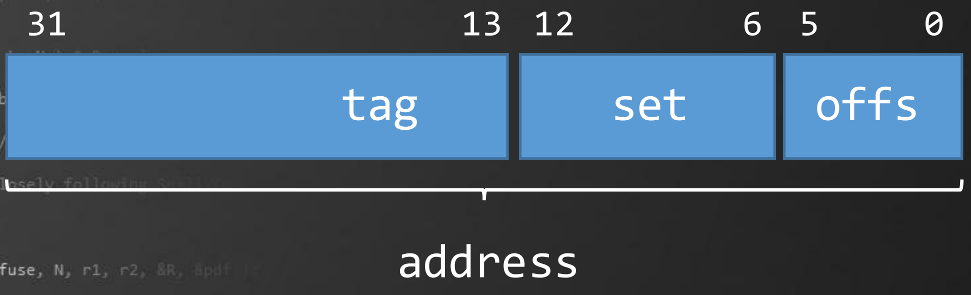
Both AMD and Intel currently use 64 byte cache lines.

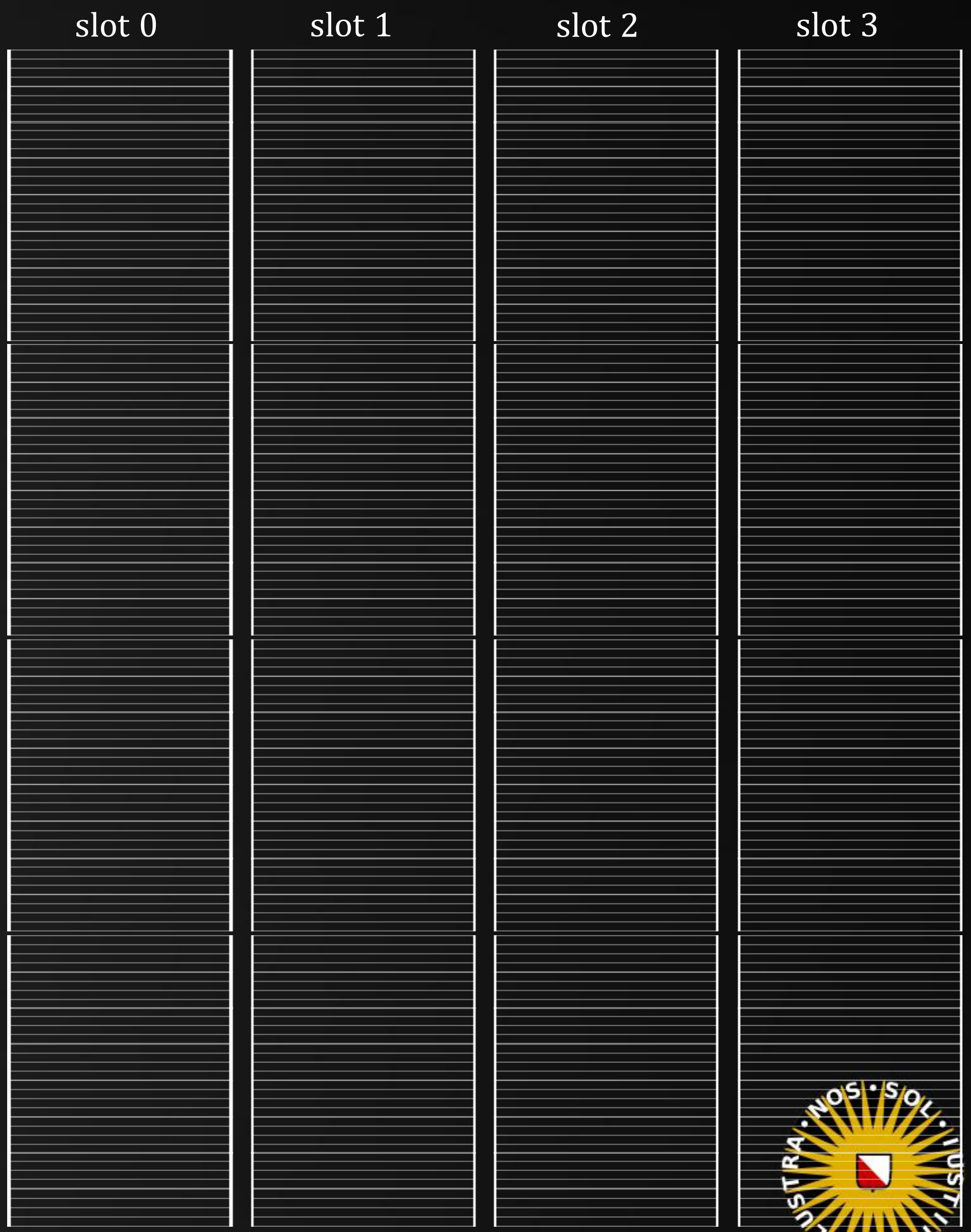
Architectures

32KB, 4-Way Set Associative Cache

```
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

/INFOMOV/

END of “Caching (1)”

next lecture: “SIMD (1)”