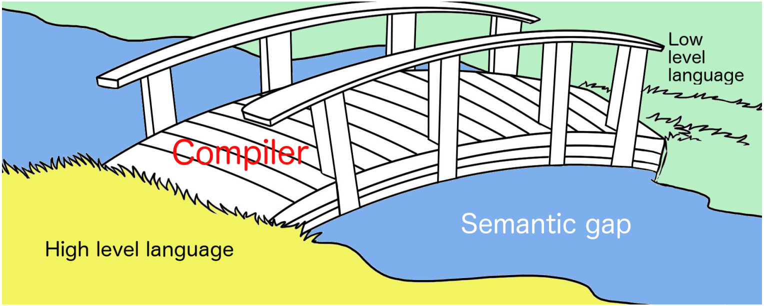
Compilers and computer architecture: Caches and caching

Martin Berger [1]

December 2019

[1] Email: `M.F.Berger@sussex.ac.uk`, Office hours: Wed 12-13 in Chi-2R312

Recall the function of compilers



Caches in modern CPUs

Today we will learn about caches in modern CPUs. They are crucial for high-performance programs and high-performance compilation.

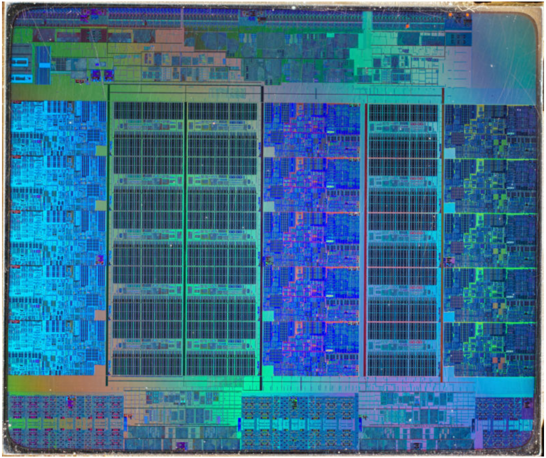
Caches in modern CPUs

Today we will learn about caches in modern CPUs. They are crucial for high-performance programs and high-performance compilation.

Today's material can safely be ignored by 'normal' programmers who don't care about performance.

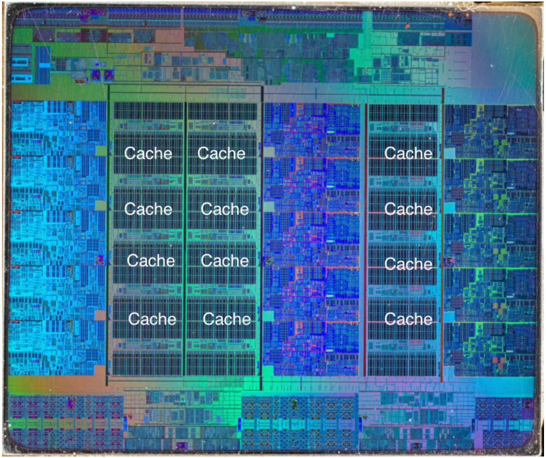
Caches in modern CPUs

Let's look at a modern CPU. Here is a November 2018 Intel Ivy Bridge Xeon CPU. Much of the silicon is for the cache, and cache controllers.

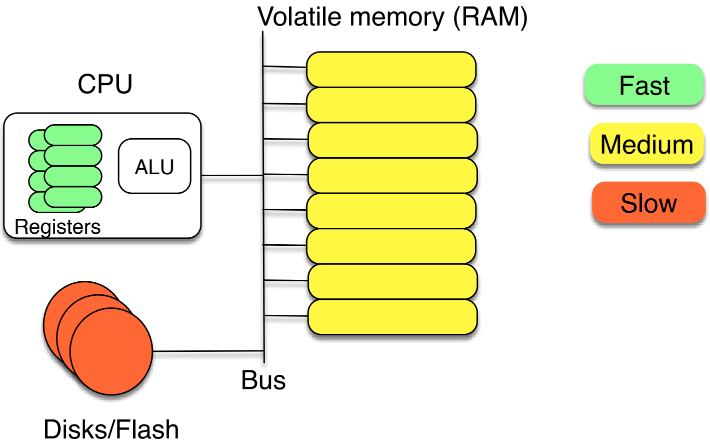


Caches in modern CPUs

Why is much of the chip area dedicated to caches?



Simplified computer layout



The faster the memory, the faster the computer.

Available memory

	Capacity	Latency	Cost
Register	1000s of bytes	1 cycle	££££££
SRAM	1s of MBytes	several cycles	££££
DRAM	10s GBytes	20 - 100 cycles	££
Flash	100s of GBytes		£
Hard disk	10 TByte	0.5 - 5 M cycles	cheap
Ideal	1000s GBytes	1 cycle	cheap

- RAM = Random Access Memory
- SRAM = static RAM, fast but uses 6 transistors per bit.
- DRAM = dynamic RAM, slow but uses 1 transistor per bit.
- Flash = non-volatile, slow, looses capacity over time.
- Latency is the time between issuing the read/write access and the completion of this access.

Available memory

	Capacity	Latency	Cost
Register	1000s of bytes	1 cycle	££££££
SRAM	1s of MBytes	several cycles	££££
DRAM	10s GBytes	20 - 100 cycles	££
Flash	100s of GBytes		£
Hard disk	10 TByte	0.5 - 5 M cycles	cheap
Ideal	1000s GBytes	1 cycle	cheap

- RAM = Random Access Memory
- SRAM = static RAM, fast but uses 6 transistors per bit.
- DRAM = dynamic RAM, slow but uses 1 transistor per bit.
- Flash = non-volatile, slow, looses capacity over time.
- Latency is the time between issuing the read/write access and the completion of this access.

It seems memory is either small, fast and expensive, or cheap, big and slow.

Tricks for making the computer run faster

Tricks for making the computer run faster

Key ideas:

- Use a **hierarchy** of memory technology.
- Keep the most often-used data in a small, fast memory.
- Use slow main memory only rarely.

Tricks for making the computer run faster

Key ideas:

- Use a **hierarchy** of memory technology.
- Keep the most often-used data in a small, fast memory.
- Use slow main memory only rarely.

But how?

Tricks for making the computer run faster

Key ideas:

- Use a **hierarchy** of memory technology.
- Keep the most often-used data in a small, fast memory.
- Use slow main memory only rarely.

But how? Does that mean the programmer should have to worry about this?

Tricks for making the computer run faster

Key ideas:

- Use a **hierarchy** of memory technology.
- Keep the most often-used data in a small, fast memory.
- Use slow main memory only rarely.

But how? Does that mean the programmer should have to worry about this? Let the CPU worry about this, using a mechanism called **cache**.

Tricks for making the computer run faster

Key ideas:

- Use a **hierarchy** of memory technology.
- Keep the most often-used data in a small, fast memory.
- Use slow main memory only rarely.

But how? Does that mean the programmer should have to worry about this? Let the CPU worry about this, using a mechanism called **cache**.

Why on earth should this work?

Tricks for making the computer run faster

Key ideas:

- Use a **hierarchy** of memory technology.
- Keep the most often-used data in a small, fast memory.
- Use slow main memory only rarely.

But how? Does that mean the programmer should have to worry about this? Let the CPU worry about this, using a mechanism called **cache**.

Why on earth should this work?

Exploit **locality!**

Locality

In practice, most programs, most of the time, do the following:

Locality

In practice, most programs, most of the time, do the following:

> *If the program accesses memory location `loc` then the most of the next few memory accesses are very likely at addresses close to `loc`.*

Locality

In practice, most programs, most of the time, do the following:

> *If the program accesses memory location `loc` then the most of the next few memory accesses are very likely at addresses close to `loc`.*

I.e. we use access to a memory location as a **heuristic prediction** for memory access in the (near) future. This is called **(data) locality**. Locality means that memory access often (but not necessarily always) follows this fairly **predictable** pattern.

Modern CPUs exploit this predictability for speed with caches.

Locality

In practice, most programs, most of the time, do the following:

> *If the program accesses memory location `loc` then the most of the next few memory accesses are very likely at addresses close to `loc`.*

I.e. we use access to a memory location as a **heuristic prediction** for memory access in the (near) future. This is called **(data) locality**. Locality means that memory access often (but not necessarily always) follows this fairly **predictable** pattern.

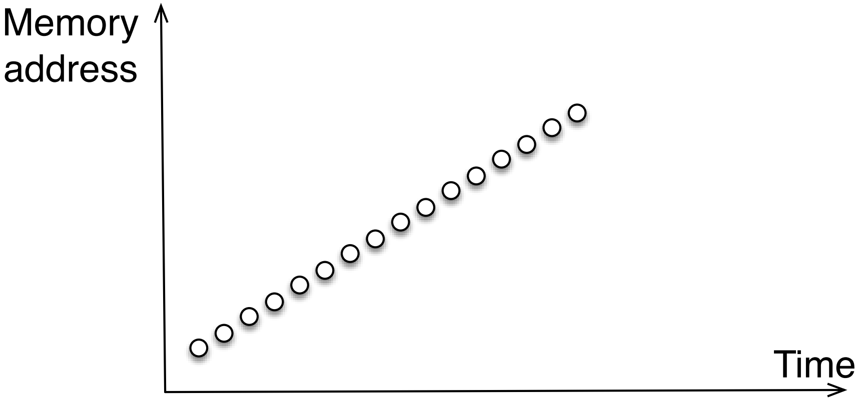
Modern CPUs exploit this predictability for speed with caches.

Note: it is possible to write programs that don't exhibit locality. They will typically run very slow.

Why would most programs exhibit locality?

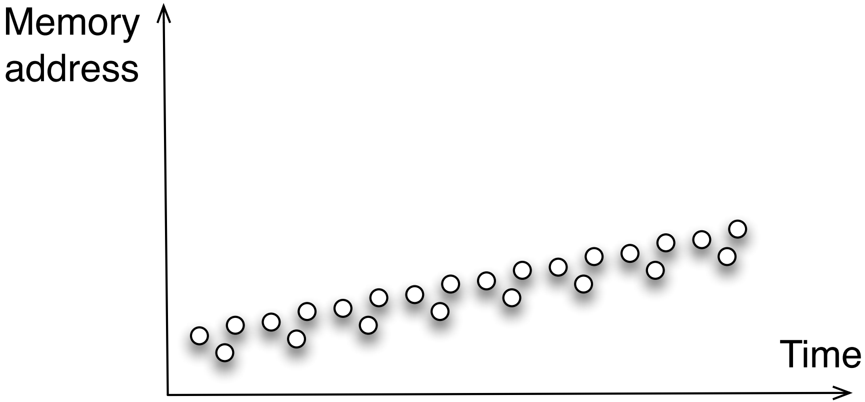
Data locality

```
int [] a = new int [1000000];
for ( int i = 0; i < 1000000; i++ ) {
   a [ i ] = i+1; }
```



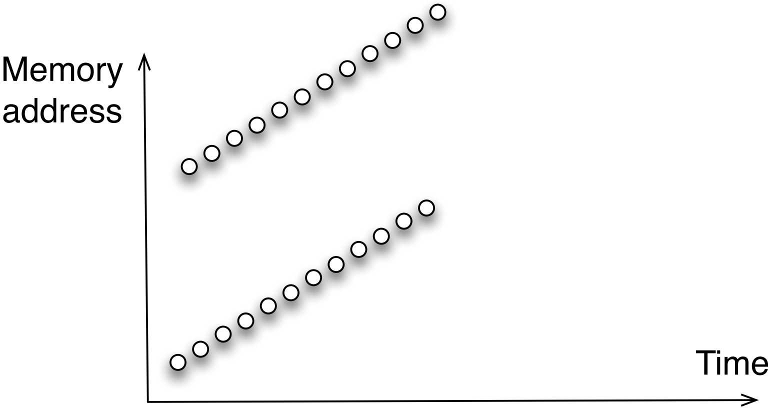
Data locality

```
int [] a = new int [1000000];
...
for ( int i = 2; i < 1000000; i++ ) {
    a [ i ] = a [i-1] * a [i-2]; }
```



Data locality

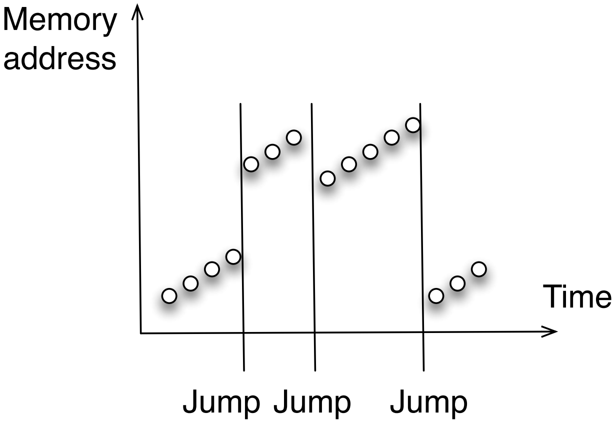
```
int [] a = new int [1000000];
int [] b = new int [1000000];
...
for ( int i = 0; i < 1000000; i++ ) {
    a [ i ] = b [ i ] + 1; }
```



Code locality

Program execution (reading via PC) is local too, with occasional jumps.

```
addiu $sp $sp
li $a0 1
lw $t1 4($sp)
sub $a0 $t1 $a
addiu $sp $sp
sw $a0 0($sp)
addiu $sp $sp
jal next
lw $t1 4($sp)
add $a0 $t1 $a
addiu $sp $sp
b exit2
```

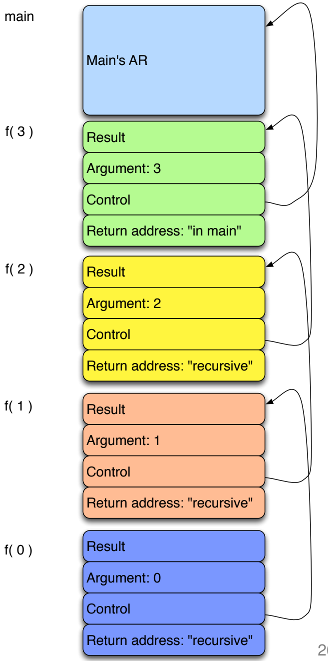


Data locality

Another cause for data locality is the stack and how we compile procedure invocations into activation records.

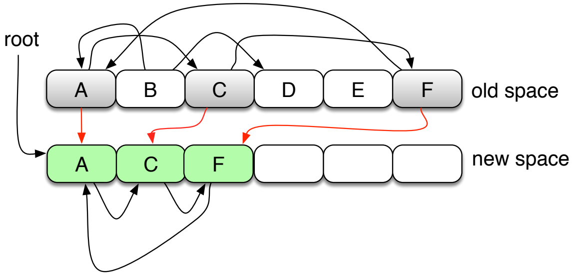
This is because within a procedure activation we typically spend a lot of time accessing the procedure arguments and local variables.

In addition, in recursive procedure invocations, related activation records, are nearby on the stack.



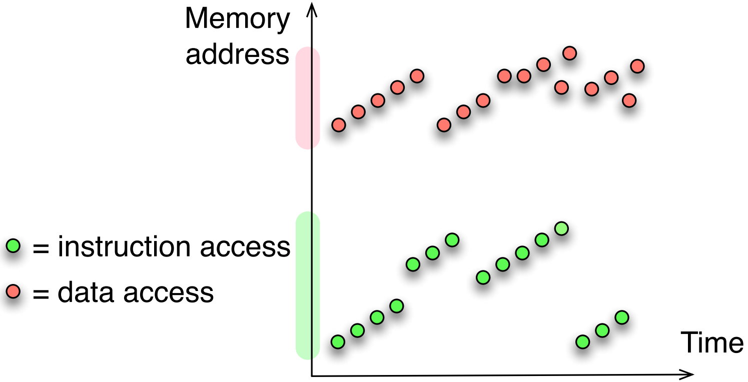
Data locality

Stop & Copy garbage collectors improve locality because they **compact** the heap.



Locality

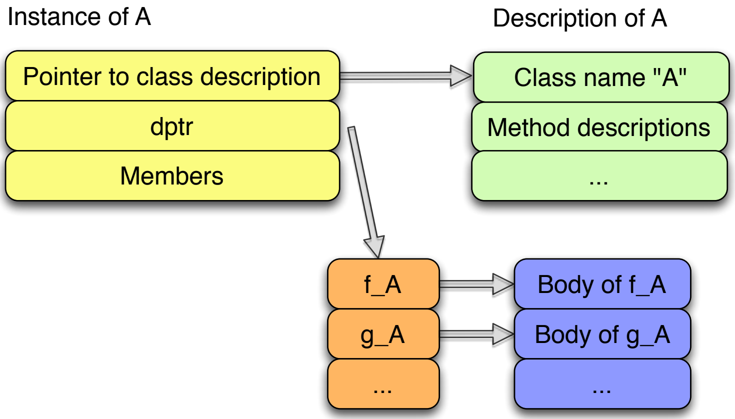
In practise we have data access and instruction access together, so the access patterns look more like this:



Still a lot of predictability in memory access patterns, but over (at least) two distinct regions of memory.

Data locality of OO programming

Accessing objects, especially method invocation often has **bad** locality because of pointer chasing. Object pointers can point anywhere inside the heap, loosing locality.



Partly to ameliorate this shortcoming, JIT compilers have been developed.

How to exploit locality and the memory hierarchy?

How to exploit locality and the memory hierarchy?

Two approaches

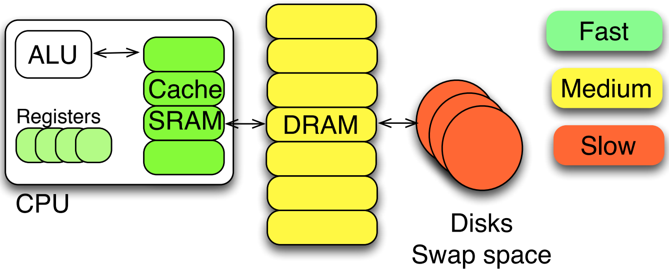
How to exploit locality and the memory hierarchy?

Two approaches

- **Expose** the hierarchy (proposed by S. Cray): let programmers access registers, fast SRAM, slow DRAM and the disk 'by hand'. Tell them "Use the hierarchy cleverly". This is not done in 2019.
- **Hide** the memory hierarchy. Programming model: there is a **single** kind of memory, single address space (excluding registers). **Automatically** assigns locations to fast or slow memory, depending on usage patterns. This is what **caches** do in CPUs.

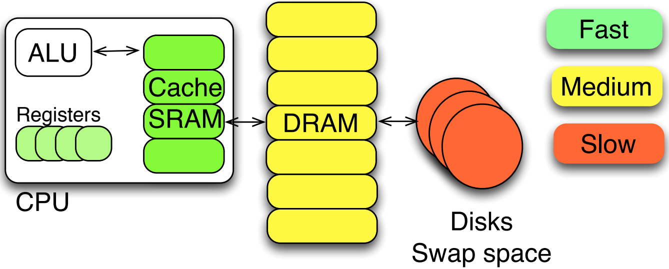
Cache

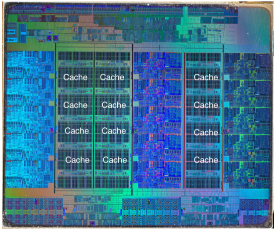
The key element is the **cache** which is integrated in modern CPUs.



Cache

The key element is the **cache** which is integrated in modern CPUs.





Cache

A CPU **cache** is used by the CPU of a computer to reduce the average time to access memory. The cache is a smaller, faster and more expensive memory inside the CPU which stores copies of the data from the **most frequently** used main memory locations for fast access.

Cache

A CPU **cache** is used by the CPU of a computer to reduce the average time to access memory. The cache is a smaller, faster and more expensive memory inside the CPU which stores copies of the data from the **most frequently** used main memory locations for fast access. When the CPU reads from or writes to a location in main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory. As long as most memory accesses are cached memory locations, the average latency of memory accesses will be closer to the cache latency than to the latency of main memory.

Cache

A CPU **cache** is used by the CPU of a computer to reduce the average time to access memory. The cache is a smaller, faster and more expensive memory inside the CPU which stores copies of the data from the **most frequently** used main memory locations for fast access. When the CPU reads from or writes to a location in main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory. As long as most memory accesses are cached memory locations, the average latency of memory accesses will be closer to the cache latency than to the latency of main memory.

Recall that **latency** (of memory access) is the time between issuing the read/write access and the completion of this access.

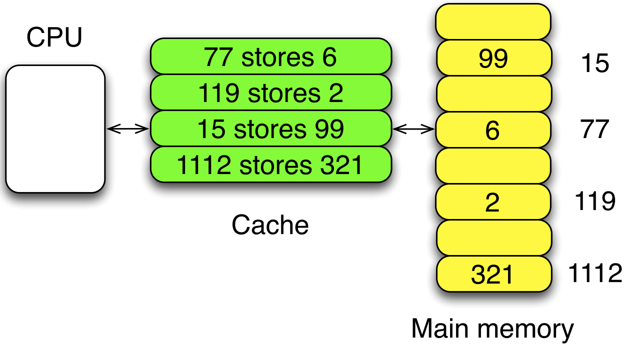
Cache

A CPU **cache** is used by the CPU of a computer to reduce the average time to access memory. The cache is a smaller, faster and more expensive memory inside the CPU which stores copies of the data from the **most frequently** used main memory locations for fast access. When the CPU reads from or writes to a location in main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory. As long as most memory accesses are cached memory locations, the average latency of memory accesses will be closer to the cache latency than to the latency of main memory.

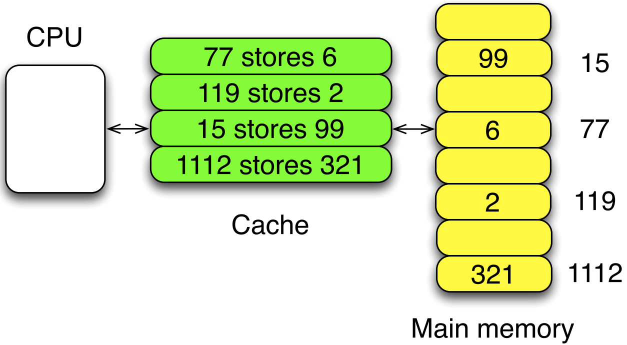
Recall that **latency** (of memory access) is the time between issuing the read/write access and the completion of this access.

A cache entry is called **cache line**.

Cache (highly simplified)

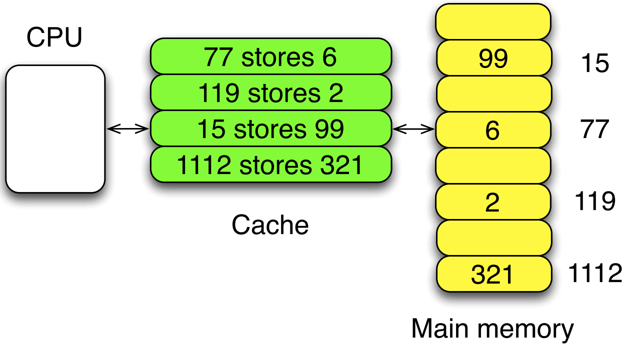


Cache (highly simplified)



Cache contains **temporary copies** of selected main memory locations, eg. Mem[119] = 2.

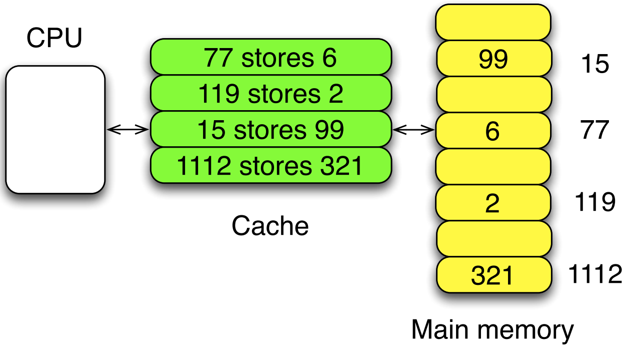
Cache (highly simplified)



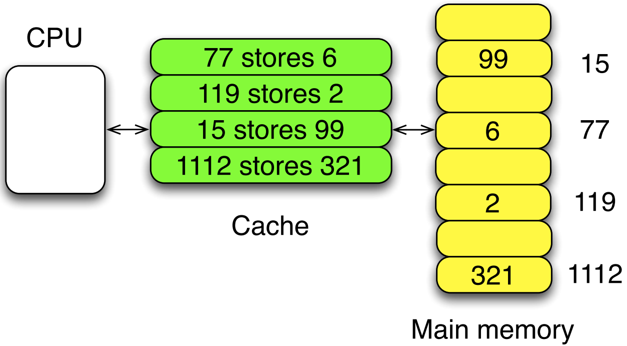
Cache contains **temporary copies** of selected main memory locations, eg. Mem[119] = 2.

The cache holds **pairs** of main memory address (called **tag**) and value.

Cache (highly simplified)

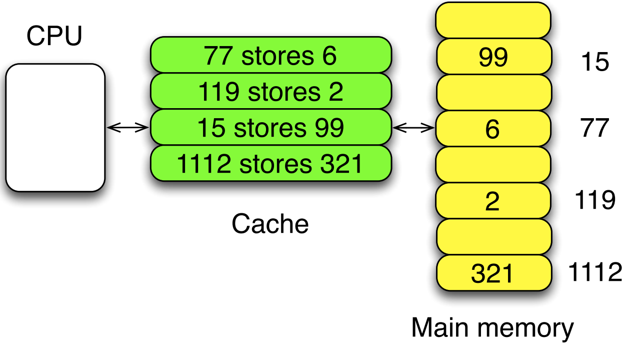


Cache (highly simplified)



Goal of a cache: to reduce the **average** access time to memory by exploiting locality.

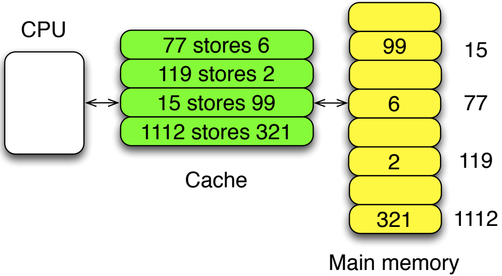
Cache (highly simplified)



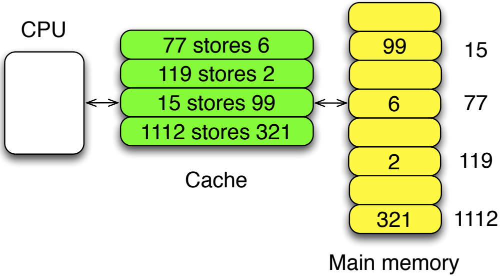
Goal of a cache: to reduce the **average** access time to memory by exploiting locality.

Caches are made from expensive but fast SRAM, with much less capacity than main memory. So not all memory entries can be in cache.

Cache reading (highly simplified)



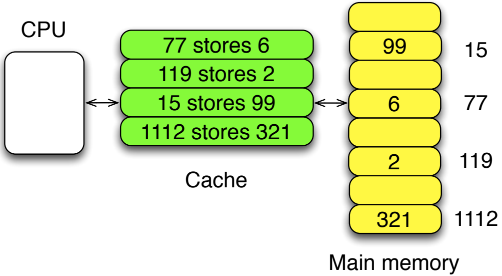
Cache reading (highly simplified)



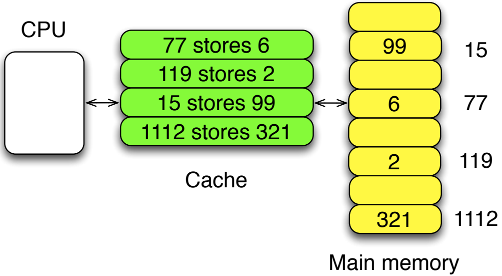
Cache 'algorithm': if the CPU wants to read memory location `loc`:

- Look for tag `loc` in cache.
- If cache line (`loc`, val) is found (called **cache hit**), then return val.
- If no cache line contains tag `loc` (called **cache miss**), then select some cache line k for replacement, read location `loc` from main memory getting value val', replace k with (`loc`, val').

Cache writing (highly simplified)



Cache writing (highly simplified)



Cache 'algorithm': if the CPU wants to write the value val memory location `loc`:

- Write val to main memory location `loc`.
- Look for tag `loc` in cache.
- If cache line (`loc`, val') is found (**cache hit**), then replace val' with val in the cache line.
- If no cache line contains tag `loc` (**cache miss**), then select some cache line k for replacement, replace k with (`loc`, val).

Note

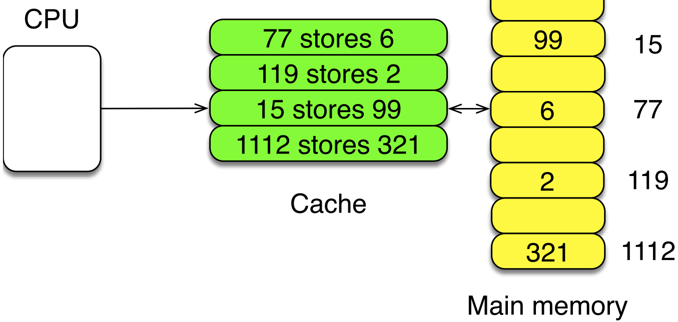
All these things (writing to main memory, looking for tag, replacing cache line, evicting etc) happen **automatically**, behind the programmer's back. It's all implemented in silicon, so cache management is **very fast**.

Note

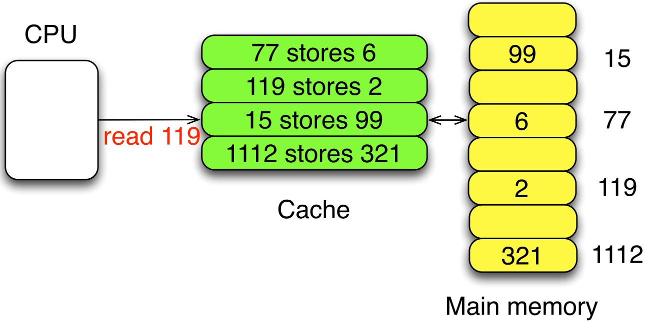
All these things (writing to main memory, looking for tag, replacing cache line, evicting etc) happen **automatically**, behind the programmer's back. It's all implemented in silicon, so cache management is **very fast**.

Unless interested in peak performance, the programmer can program under the illusion of memory uniformity.

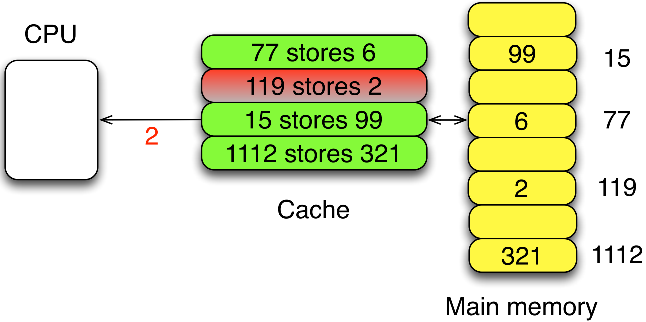
Successful cache read (highly simplified)



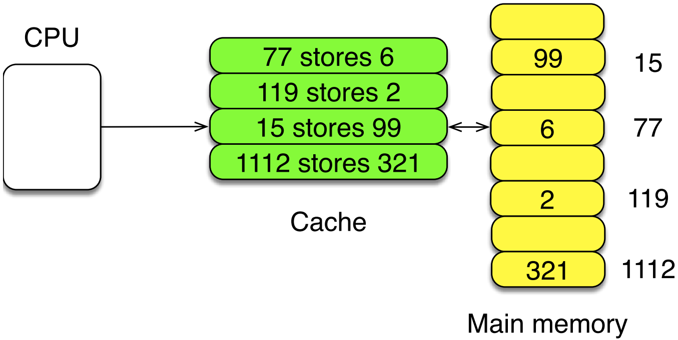
Successful cache read (highly simplified)



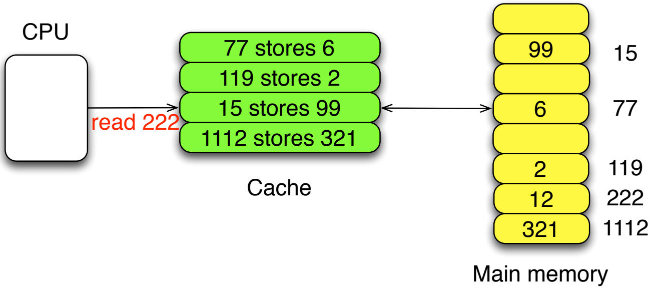
Successful cache read (highly simplified)



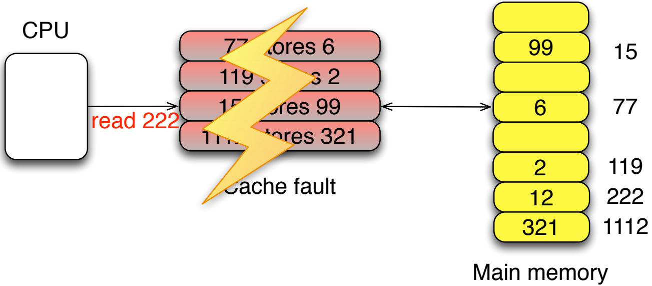
Successful cache read (highly simplified)



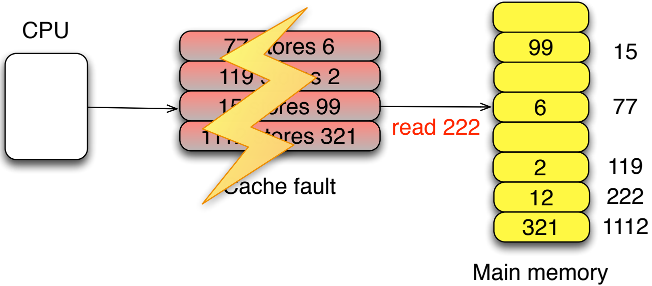
Cache failure on read (highly simplified)



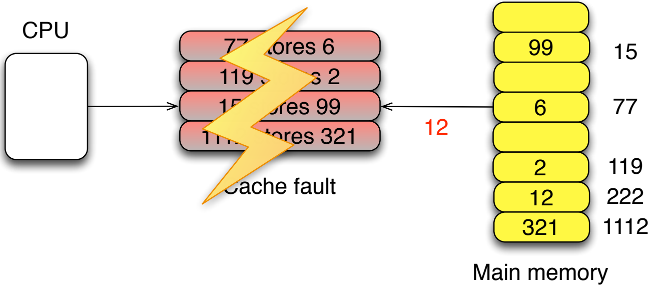
Cache failure on read (highly simplified)



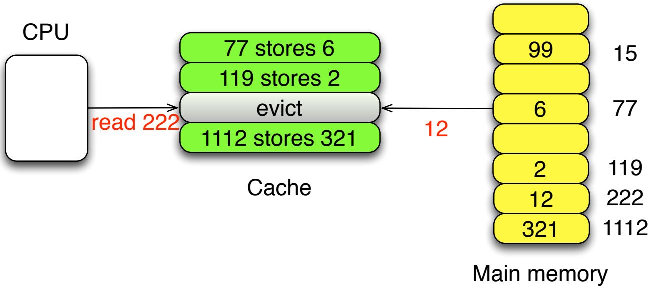
Cache failure on read (highly simplified)



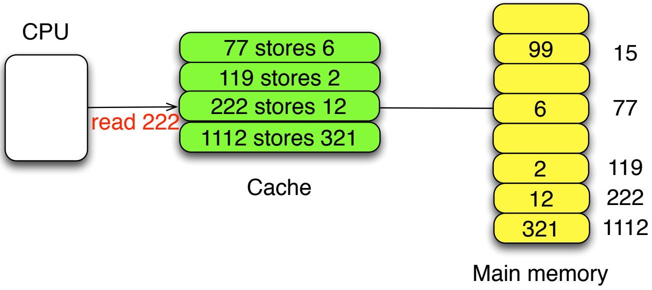
Cache failure on read (highly simplified)



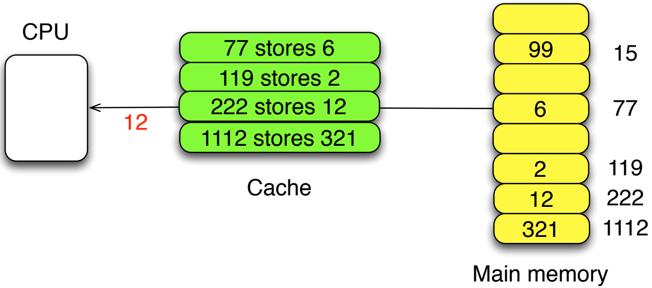
Cache failure on read (highly simplified)



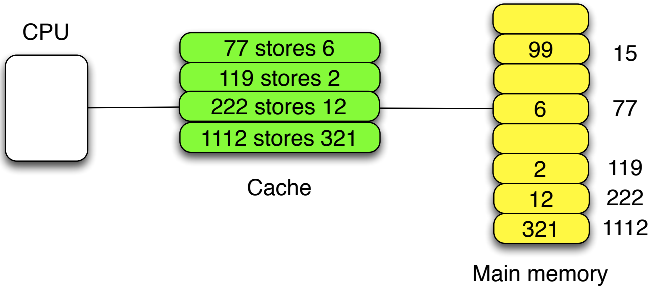
Cache failure on read (highly simplified)



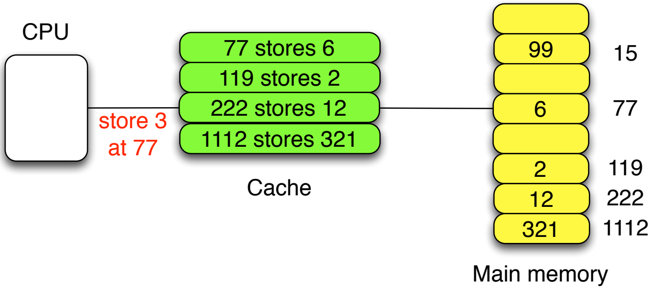
Cache failure on read (highly simplified)



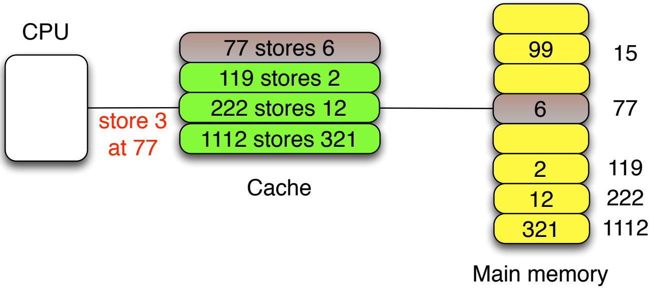
Cache failure on read (highly simplified)



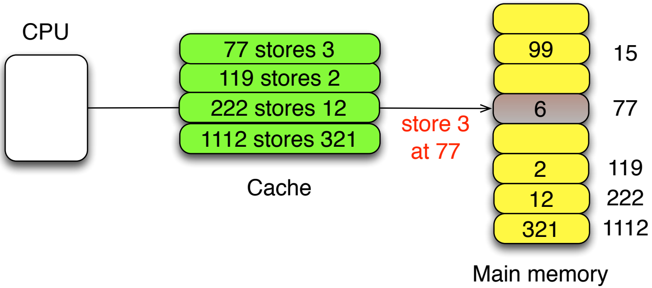
Successful cache write (highly simplified)



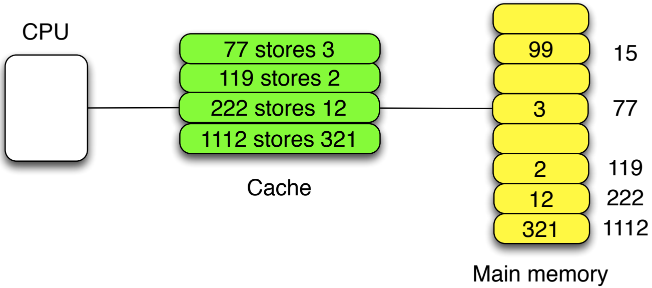
Successful cache write (highly simplified)



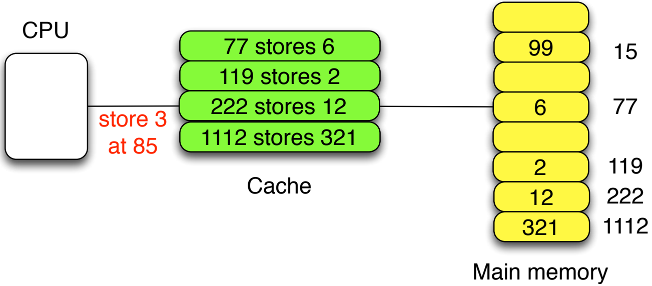
Successful cache write (highly simplified)



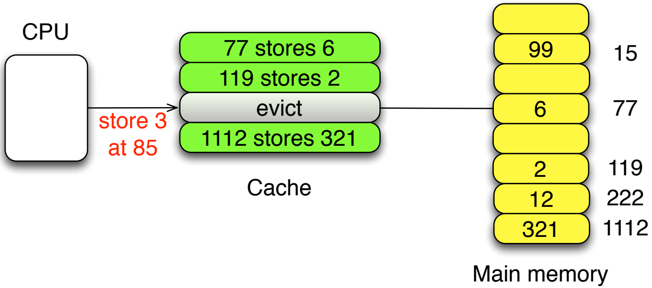
Successful cache write (highly simplified)



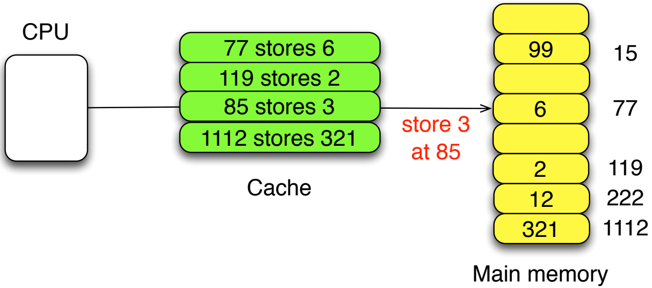
Write with cache failure (highly simplified)



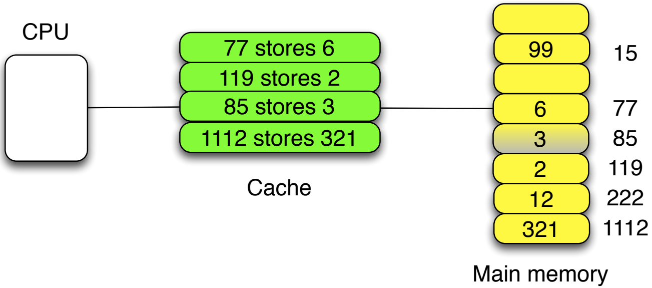
Write with cache failure (highly simplified)



Write with cache failure (highly simplified)



Write with cache failure (highly simplified)



Prefetching

What about locality?

Prefetching

What about locality? Explanation has so far made little use of locality.

Prefetching

What about locality? Explanation has so far made little use of locality.

Caches typically do something else. By locality, the act of

accessing `loc` predicts likely usage of

```
... loc-1    loc    loc+1    loc+2    loc+3    ...
```

in the near future. So far we are only ’reusing’ `loc`.

Prefetching

What about locality? Explanation has so far made little use of locality.

Caches typically do something else. By locality, the act of

accessing `loc` predicts likely usage of

```
... loc-1    loc    loc+1    loc+2    loc+3    ...
```

in the near future. So far we are only 'reusing' `loc`.

So let's get e.g. `loc...loc+n` from memory together (rather than separately). This is called **prefetching** . It relies on the fact that with current CPUs and memory systems a burst of reads of *n* adjacent addresses from main memory is faster than fetching those *n* cells separately.

Prefetching

So on cache fault on `loc`, the CPU prefetches e.g. `loc...loc+n` in one big burst.

Prefetching

So on cache fault on `loc`, the CPU prefetches
e.g. `loc...loc+n` in one big burst.

The concept of **preloading** of webpages in is similar.
(Cf. Instagram)

Counterintuitive behaviour of CPUs with caches

Consider the following simple program.

```
x := x + 1;
x := x + 1
```

Clearly both assignments translate to the same machine code. But if the CPU has caches (and all modern CPUs have), then the execution of the first execution of $\mathtt{x} := \mathtt{x} + 1$ might take much longer than the execution of the second $\mathtt{x} := \mathtt{x} + 1$, despite having identical machine code. Why?

Counterintuitive behaviour of CPUs with caches

Consider the following simple program.

```
x := x + 1;
x := x + 1
```

Clearly both assignments translate to the same machine code. But if the CPU has caches (and all modern CPUs have), then the execution of the first execution of $\mathtt{x} := \mathtt{x} + 1$ might take much longer than the execution of the second $\mathtt{x} := \mathtt{x} + 1$, despite having identical machine code. Why?

Because in the first execution of $\mathtt{x} := \mathtt{x} + 1$, the cache may not contain $\mathtt{x}$. In this case, a slow request will be issued to main memory to fetching $\mathtt{x}$. However, if for some reason $\mathtt{x}$ is already in the cache then $\mathtt{x}$ will quickly be fetched from the cache. In both cases the second execution will execute quickly, because the cache will contain $\mathtt{x}$.

Miss rate

$$\text{Miss rate} = \frac{\text{number of cache failures}}{\text{number of cache accesses}}.$$

Miss rate

$$\text{Miss rate} = \frac{\text{number of cache failures}}{\text{number of cache accesses}}.$$

The performance of a program can be significantly improved if the miss rate is low.

Miss rate

Miss rate = $\frac{\text{number of cache failures}}{\text{number of cache accesses}}$.

The performance of a program can be significantly improved if the miss rate is low.

Miss rate is influenced by many factors:

Miss rate

Miss rate = $\frac{\text{number of cache failures}}{\text{number of cache accesses}}$.

The performance of a program can be significantly improved if the miss rate is low.

Miss rate is influenced by many factors:

- Selection policy for cache eviction.
- Compiler improving data locality (e.g. in garbage collection).
- Programmer ensuring data locality.
- Cache size (bigger = better). Modern computers have multiple caches (see later).

Miss penalty

The **miss penalty** of a cache are the time it takes to read from/write to main memory - (minus) the time it takes to read from/write to the cache.

Miss penalty

The **miss penalty** of a cache are the time it takes to read from/write to main memory - (minus) the time it takes to read from/write to the cache.

The miss penalty has been increasing dramatically, because CPUs get faster more quickly than memory. So good cache management is becoming increasingly important.

Important questions about Cache

Important questions about Cache

How does the search in the cache for a tag proceed?

Important questions about Cache

How does the search in the cache for a tag proceed?

Which cache line is replaced after a cache miss?

Important questions about Cache

How does the search in the cache for a tag proceed?

Which cache line is replaced after a cache miss?

When to update main memory after cache write?

Important questions about Cache

How does the search in the cache for a tag proceed?

Which cache line is replaced after a cache miss?

When to update main memory after cache write?

What (how much) data to read from main memory after a cache miss?

Important questions about Cache

How does the search in the cache for a tag proceed?

Which cache line is replaced after a cache miss?

When to update main memory after cache write?

What (how much) data to read from main memory after a cache miss?

The answers to both are vitally important for performance. Different CPUs give different answers. Modern CPUs are **very** sophisticated in these matters: good answers have dramatic impact on performance.

Important questions about Cache

Important questions about Cache

Example: which cache line is replaced after a cache miss?

Important questions about Cache

Example: which cache line is replaced after a cache miss?
Different policies:

- Random choice.
- Least recently used.
- Most recently used.
- FIFO.
- LIFO
- ...

See e.g. `https://en.wikipedia.org/wiki/Cache_replacement_policies` for more.

Important questions about Cache

Important questions about Cache

Example: when to update main memory after cache write?

Important questions about Cache

Example: when to update main memory after cache write?

Different policies:

- Immediately on write. This is simple and simple to implement, but can potentially stall the CPU while being executed.
- Later, e.g. when there is no/little contention for memory bus. This leads to strange effects from the programmer's point of view, where seemingly instructions are **reordered**, e.g. the programmer says: write this, then read that, but the CPU executes: read that then write this. This reordering of instructions is called the CPU's **memory model**. Modern CPUs have complicated memory models. Compiler writes and assembler programmers need to be aware of this, otherwise programs will be buggy. 'Normal' programmers can ignore this, since the compiler will sort it out.

Important questions about Cache

Important questions about Cache

Example: what (how much) data to read from main memory after a cache miss?

Important questions about Cache

Example: what (how much) data to read from main memory after a cache miss?

Different CPUs give different answers. Due to locality, it makes sense to load `loc+0`, ..., `loc+n` on cache fault for location `loc`. Modern CPUs typically determine *n* **dynamically**.

Important questions about Cache

Important questions about Cache

What do real processors use?

Important questions about Cache

What do real processors use? That's quite hard to say, but looks complicated. CPU producers seem to keep this secret. If you know more, let me know ...

Important questions about Cache

What do real processors use? That's quite hard to say, but looks complicated. CPU producers seem to keep this secret. If you know more, let me know ...

Why can they keep this secret?

Important questions about Cache

What do real processors use? That's quite hard to say, but looks complicated. CPU producers seem to keep this secret. If you know more, let me know ...

Why can they keep this secret? Because caching does not affect the results programs compute, only speed. Give manufacturers freedom to change CPU architecture without telling anyone.

An artificial example

```
for ( j <- 0 to 20-1 ) {
   for ( i <- 0 to 100000000-1 ) {
      a[i] = a[i]*b[i] } }
```

When we run this we execute

```
a[0] = a[0] * b[0]
a[1] = a[1] * b[1]
a[2] = a[2] * b[2]
a[3] = a[3] * b[3]
a[4] = a[4] * b[4]
...
a[0] = a[0] * b[0]
a[1] = a[1] * b[1]
...
```

An artificial example

```
for ( j <- 0 to 20-1 ) {
   for ( i <- 0 to 100000000-1 ) {
      a[i] = a[i]*b[i] } }
```

When we run this we execute

```
a[0] = a[0] * b[0]
a[1] = a[1] * b[1]
a[2] = a[2] * b[2]
a[3] = a[3] * b[3]
a[4] = a[4] * b[4]
...
a[0] = a[0] * b[0]
a[1] = a[1] * b[1]
...
```

Depending on details, every read is a cache miss. So the program runs at the speed of main memory, i.e. slowly.

An artificial example

What happens if we exchange loops?

```
for ( i <- 0 to 100000000-1 ) {
   for ( j <- 0 to 20-1 ) {
      a[i] = a[i]*b[i] } }
```

When we run this we execute

```
a[0] = a[0] * b[0]
a[0] = a[0] * b[0]
...
a[0] = a[0] * b[0]
a[1] = a[1] * b[1]
...
a[1] = a[1] * b[1]
...
```

An artificial example

What happens if we exchange loops?

```
for ( i <- 0 to 100000000-1 ) {
   for ( j <- 0 to 20-1 ) {
      a[i] = a[i]*b[i] } }
```

When we run this we execute

```
a[0] = a[0] * b[0]
a[0] = a[0] * b[0]
...
a[0] = a[0] * b[0]
a[1] = a[1] * b[1]
...
a[1] = a[1] * b[1]
...
```

Result:

- The program computes exactly the same results.
- We have a lot fewer cache misses.

An artificial example

Going from

```
for ( j <- 0 to 20-1 ) {
  for ( i <- 0 to 100000000-1 ) {
    a[i] = a[i]*b[i] } }
```

to

```
for ( i <- 0 to 100000000-1 ) {
  for ( j <- 0 to 20-1 ) {
    a[i] = a[i]*b[i] } }
```

is called **loop interchange**, can speed up code up to 10 times, and some advanced compilers can do it automatically.

See `prog/proc.c`

Problem

Modern compilers are very good at managing registers, much better than almost all programmers could.

Problem

Modern compilers are very good at managing registers, much better than almost all programmers could.

Compilers are **not** good at managing caches. This problem is left to the programmer. It is an open research question of to make compilers better at managing caches.

Problem

Modern compilers are very good at managing registers, much better than almost all programmers could.

Compilers are **not** good at managing caches. This problem is left to the programmer. It is an open research question of to make compilers better at managing caches.

One problem: OS switches between processes, and a context switch typically 'trashes' the cache, i.e. the cache holds values that are good for the outgoing process, but hold no values of interest to the incoming process.

Multi-level caches

Multi-level caches

The picture painted about caches so far is too simplistic in that modern CPUs have not one but (usually) three caches:

- L1. Small, very fast.
- L2. Medium size, medium speed.
- L3. Large size, slow (but still much faster than main memory).

(L stands for Level.) Intel has just started adding L4 caches (Haswell)

Multi-level caches

The picture painted about caches so far is too simplistic in that modern CPUs have not one but (usually) three caches:

- L1. Small, very fast.
- L2. Medium size, medium speed.
- L3. Large size, slow (but still much faster than main memory).

(L stands for Level.) Intel has just started adding L4 caches (Haswell)

Processor	L1	L2	L3
ARM Coretex A9	32 KB	128kB-8MB	-
ARM Coretex A15	64 KB	4MB	-
Intel Ivy Bridge	64 KB p. core	256 kB p. core	8 MB shared

Multi-level caches

The picture painted about caches so far is too simplistic in that modern CPUs have not one but (usually) three caches:

- L1. Small, very fast.
- L2. Medium size, medium speed.
- L3. Large size, slow (but still much faster than main memory).

(L stands for Level.) Intel has just started adding L4 caches (Haswell)

Processor	L1	L2	L3
ARM Coretex A9	32 KB	128kB-8MB	-
ARM Coretex A15	64 KB	4MB	-
Intel Ivy Bridge	64 KB p. core	256 kB p. core	8 MB shared

Why?

Multi-level caches

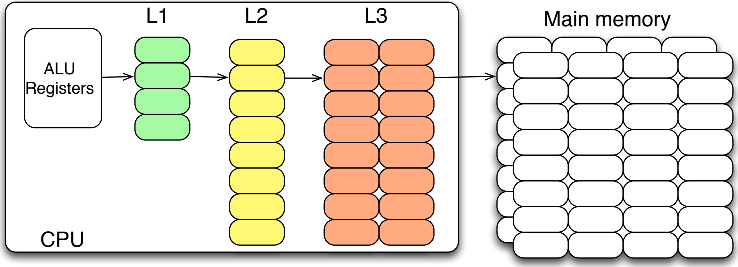
There is a fundamenal and hard trade-off in caches:

- Smaller chaches are faster, have lower latency.
- Bigger caches have a better miss-rate.

It seems we can't have both: fast caches and low miss-rate.

Multi-level caches

To deal with the miss-rate/low latency trade-off, modern CPU create a **hierarchy** of caches: the small but fast L1 cache doesn't read directly from memory but from a bigger but slower L2 cache. In turn the L2 cache often reads from a even larger and even more slow L3 cache. The L3 cache reads from the main memory.

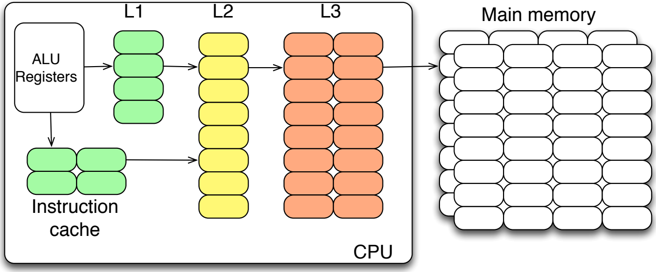


Instruction caches

So far we've mostly assumed that our caches can not only be read from, but also written to. This is vital for data.

But in most modern computers, instructions can only be read. Caches that are read-only are much easier technically and hence faster, and taking up less chip space.

Consequently, modern CPUs often have a separate and fast **instruction cache** that exploits instruction locality.



Interesting issues not mentioned

Interesting issues not mentioned

- Modern CPUs have multiple cores, i.e. mini-CPUs. How do cores and caches interact?

Interesting issues not mentioned

- Modern CPUs have multiple cores, i.e. mini-CPUs. How do cores and caches interact?
- Caches can be used to steal data from other processes (e.g. private keys), how can that be avoided? See e.g. the paper “CACHE MISSING FOR FUN AND PROFIT” by Colin Percival.

Interesting issues not mentioned

- Modern CPUs have multiple cores, i.e. mini-CPUs. How do cores and caches interact?
- Caches can be used to steal data from other processes (e.g. private keys), how can that be avoided? See e.g. the paper “CACHE MISSING FOR FUN AND PROFIT” by Colin Percival. Short summary: this is a serious problem, and Intel has changed it’s CPU architectures so that caches can be ’switched off’ when dealing with secret data.

Conclusion

Conclusion

Caches in modern CPUs are a form of fast memory that automatically exploits memory locality to speed up execution, often quite considerably.

Conclusion

Caches in modern CPUs are a form of fast memory that automatically exploits memory locality to speed up execution, often quite considerably.

Making good use of the cache typically has drastic influences on execution speed.

Conclusion

Caches in modern CPUs are a form of fast memory that automatically exploits memory locality to speed up execution, often quite considerably.

Making good use of the cache typically has drastic influences on execution speed.

It is currently difficult for compilers to increase data-locality (although Stop & Copy garbage collectors help).

Conclusion

Caches in modern CPUs are a form of fast memory that automatically exploits memory locality to speed up execution, often quite considerably.

Making good use of the cache typically has drastic influences on execution speed.

It is currently difficult for compilers to increase data-locality (although Stop & Copy garbage collectors help).

For most normal programming activities, it's probably not a good idea to worry much about data locality. It's probably more economical to think about better high-level algorithms (or use a faster computer) when you encounter performance problems.

Conclusion

Caches in modern CPUs are a form of fast memory that automatically exploits memory locality to speed up execution, often quite considerably.

Making good use of the cache typically has drastic influences on execution speed.

It is currently difficult for compilers to increase data-locality (although Stop & Copy garbage collectors help).

For most normal programming activities, it's probably not a good idea to worry much about data locality. It's probably more economical to think about better high-level algorithms (or use a faster computer) when you encounter performance problems.

For really high-performance computing, programming cache-aware is vital, and has a substantial (negative) influence on program structure and portability.