Optimizing Memory Usage on modern computers

Sébastien Ponce
sebastien.ponce@cern.ch

CERN

November 26^{th} 2018

Foreword

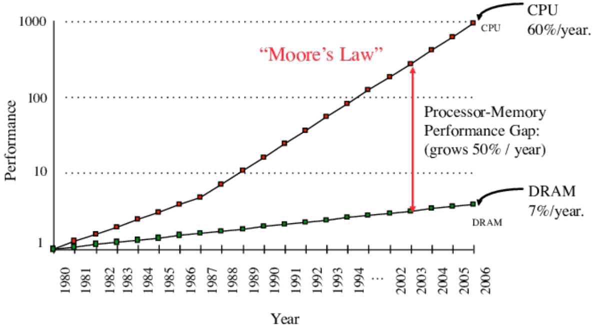
- the concepts and techniques presented here are generic
- they apply to basically all languages and data structures
- however, examples shown are based on C^{++} and the STL

Outline

1. Context
   - What is the problem with memory ?
2. How to be efficient
   - Basics of memory allocation
   - Efficient memory allocation
   - Cache optimization, SoA
3. Measuring efficiency
   - Detection of suboptimal allocations

What is the problem with memory ?

Evolution of memory in the past decades

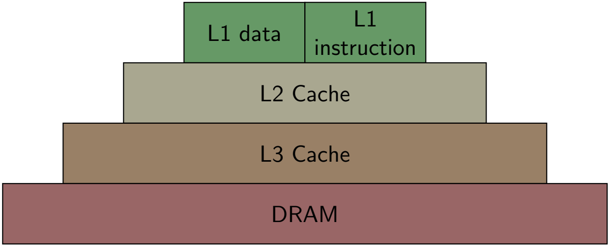


Due to Moore's law in the 80s and 90s, there is a gap between CPU and memory performances

Consequences :

- access to memory is now extremely slow (relatively)
- level of caches have been introduced to mitigate
- good usage of caches has become a key parameter

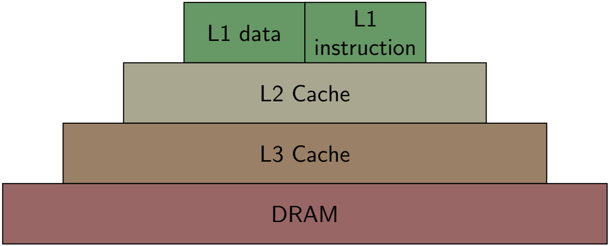
Typical cache structure



size	latency
64 kB	4 cycles
256 kB	10 cycles
10 MB	40 cycles
64 GB	400 cycles

Typical data, on an Haswell architecture

Typical cache structure



size	latency
64 kB	4 cycles
256 kB	10 cycles
10 MB	40 cycles
64 GB	400 cycles

Typical data, on an Haswell architecture

Basics of memory allocation

Process memory organization

4 main areas

- **the code segment** for the code of the executable
- **the data segment** for global variables
- **the heap** for dynamically allocated variables
- **the stack** for parameters of functions and local variables

Memory layout

heap
$\downarrow$ ... $\downarrow$
$\uparrow$ ... $\uparrow$
stack
data segment
code segment

Process memory organization

Stack allocation usage and cost

- small objects
- lifetime limited to current scope
- allocation is almost free : one CPU cyle

Heap allocation usage and cost

- any size
- lifetime infinite, until explicit deallocation
- allocation is costly :
    - find an empty piece of memory
    - going though a list/map hold by the linux kernel
    - and taking a lock to make it thread safe

Basic container in memory

Simple vector / array case

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	$\cdots$

Basic container in memory

Simple vector / array case

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	$\cdots$

Array / Vector of objects

```
struct A { float x, y, z; };
```

x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2	x_2	$\cdots$

A_0 A_1 A_2

Container of pointers

Naïve view

```
struct A { float x, y, z; };
std::vector<A*> v;   std::array<A*> a;
```

ptr_0	ptr_1	ptr_2	ptr_3	ptr_4	ptr_5	ptr_6	ptr_7	ptr_8	ptr_9	...

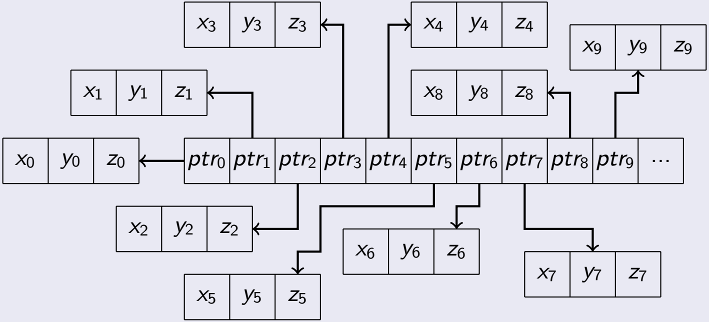
Container of pointers

Naïve view

```
struct A { float x, y, z; };
std::vector<A*> v;   std::array<A*> a;
```

ptr_0	ptr_1	ptr_2	ptr_3	ptr_4	ptr_5	ptr_6	ptr_7	ptr_8	ptr_9	...

Realistic view



Consequences : memory allocations

Optimal number of allocations

- Container of A $\rightarrow$ optimally 1 allocation, possibly on stack
- Container of A* $\rightarrow$ minimum n+1 allocations, n on heap

More consequences : reading data

Memory view for container of objects

Each line corresponds to a cache line (64 bytes, 16 floats)

0x00C0	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
0x0080	x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2	x_3	y_3	z_3	x_4	y_4	z_4	x_5
0x0040	y_5	z_5	x_6	y_6	z_6	x_7	y_7	z_7	x_8	y_8	z_8	x_9	y_9	z_9	·	·
0x0000	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·

More consequences : reading data

Memory view for container of objects

Each line corresponds to a cache line (64 bytes, 16 floats)

0x00C0	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
0x0080	x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2	x_3	y_3	z_3	x_4	y_4	z_4	x_5
0x0040	y_5	z_5	x_6	y_6	z_6	x_7	y_7	z_7	x_8	y_8	z_8	x_9	y_9	z_9	·	·
0x0000	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·

One read from RAM to Level 1 Cache is enough (2 lines in one go)

More consequences : reading data

Memory view for Container of pointers to objects

Each line corresponds to a cache line (64 bytes, 16 floats)

0x0240																
0x0200	x_8	y_8	z_8								x_9	y_9	z_9			
0x01C0						x_7	y_7	z_7								
0x0180								x_5	y_5	z_5				x_6	y_6	z_6
0x0140																
0x0100		x_3	y_3	z_3									x_4	y_4	z_4	
0x00C0						x_2	y_2	z_2								
0x0080					x_0	y_0	z_0	x_1	y_1	z_1						
0x0040	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9						
0x0000																

More consequences : reading data

Memory view for Container of pointers to objects

Each line corresponds to a cache line (64 bytes, 16 floats)

0x0240																
0x0200	x_8	y_8	z_8								x_9	y_9	z_9			
0x01C0						x_7	y_7	z_7								
0x0180								x_5	y_5	z_5				x_6	y_6	z_6
0x0140																
0x0100		x_3	y_3	z_3									x_4	y_4	z_4	
0x00C0						x_2	y_2	z_2								
0x0080					x_0	y_0	z_0	x_1	y_1	z_1						
0x0040	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9						
0x0000																

You need to read many lines, in several accesses
Remember each RAM access is 400 cycles...

Practical consequences

Guidelines

- we want as few heap memory allocations as possible
  - stack usage is much better !
- we want continuous memory blocks, allocated in one go
  - that means containers of objects, no pointers involved

Efficient memory allocation

How does a dynamic container grow ?

```
struct A { float x, y, z; };
std::vector<A> v;
```

How does a dynamic container grow ?

```
struct A { float x, y, z; };
std::vector<A> v;
```

Construction

Initially, container is empty, no storage allocated

start	0x0
finish	0x0
end_of_storage	0x0

How does a dynamic container grow ?

```cpp
struct A { float x, y, z; };
std::vector<A> v;
```

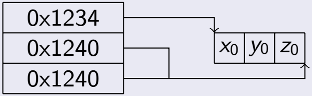
Construction

Initially, container is empty, no storage allocated

start	0x0
finish	0x0
end_of_storage	0x0

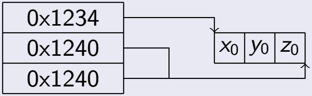
Adding first element

for std::vector, allocates storage for the first element only !



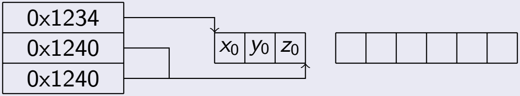
How does a dynamic container grow ?

Adding second element



How does a dynamic container grow ?

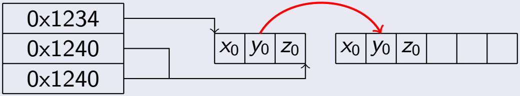
Adding second element



1. allocate new piece of memory for 2 items

How does a dynamic container grow ?

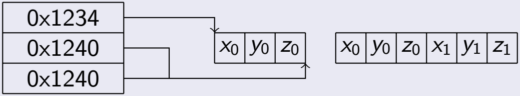
Adding second element



1. allocate new piece of memory for 2 items
2. copy existing content

How does a dynamic container grow ?

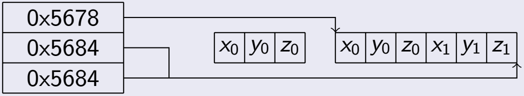
Adding second element



1. allocate new piece of memory for 2 items
2. copy existing content
3. write new content

How does a dynamic container grow ?

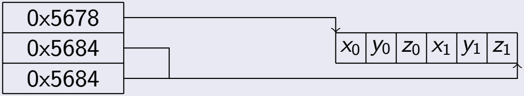
Adding second element



1. allocate new piece of memory for 2 items
2. copy existing content
3. write new content
4. update pointers

How does a dynamic container grow ?

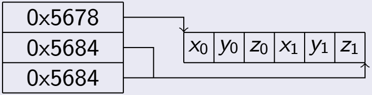
Adding second element



1. allocate new piece of memory for 2 items
2. copy existing content
3. write new content
4. update pointers
5. Deallocate original piece of memory

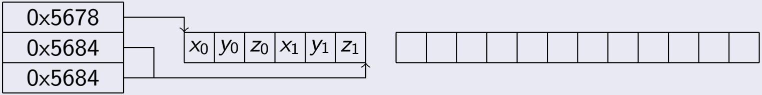
How does a dynamic container grow ?

Adding third element



How does a dynamic container grow ?

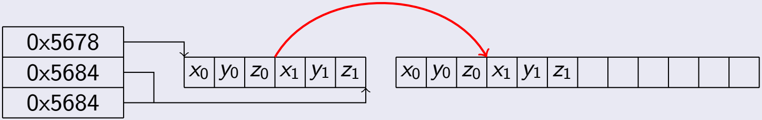
Adding third element



1. allocate new piece of memory for 4 items
   - double size at each iteration

How does a dynamic container grow ?

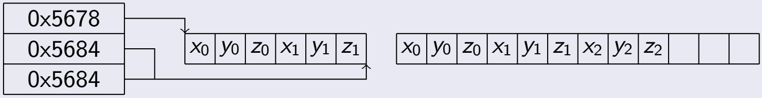
Adding third element



1. allocate new piece of memory for 4 items
   - double size at each iteration
2. copy existing content

How does a dynamic container grow ?

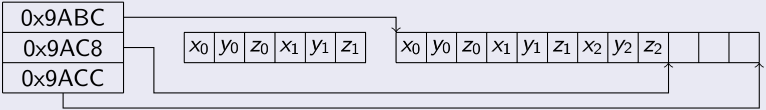
Adding third element



1. allocate new piece of memory for 4 items
   - double size at each iteration
2. copy existing content
3. write new content

How does a dynamic container grow ?

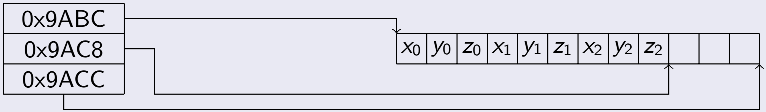
Adding third element



1. allocate new piece of memory for 4 items
   - double size at each iteration
2. copy existing content
3. write new content
4. update pointers

How does a dynamic container grow ?

Adding third element



1. allocate new piece of memory for 4 items
   - double size at each iteration
2. copy existing content
3. write new content
4. update pointers
5. Deallocate original piece of memory

Proper container allocation

You may want to avoid default behavior

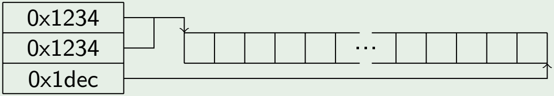
- content of container is reallocated and copied when they grow
- first item of a 1000 nodes vector is copied 10 times in C^{++} !
- when reaching 1000 items, you will have copied 1023 items in total and allocated 11 pieces of memory, releasing 10

You want to control the allocation

- and "reserve" the space manually at the start

```
std::vector<int> v;
v.reserve(1000);
```

- ensures single allocation, no copies, no reallocations



Main lessons

- use stack as much as possible, avoid heap when feasible
- use container of objects, not of pointers
- use container reservation

Cache optimization, SoA

Back to cache considerations

Memory view for container of objects

```
struct A { float a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p; }
std::vector<A> v;
```

Each `A` corresponds to a cache line (64 bytes, 16 floats)

0x01C0	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
0x0180	a_5	b_5	c_5	d_5	e_5	f_5	g_5	h_5	i_5	j_5	k_5	l_5	m_5	n_5	o_5	p_5
0x0140	a_4	b_4	c_4	d_4	e_4	f_4	g_4	h_4	i_4	j_4	k_4	l_4	m_4	n_4	o_4	p_4
0x0100	a_3	b_3	c_3	d_3	e_3	f_3	g_3	h_3	i_3	j_3	k_3	l_3	m_3	n_3	o_3	p_3
0x00C0	a_2	b_2	c_2	d_2	e_2	f_2	g_2	h_2	i_2	j_2	k_2	l_2	m_2	n_2	o_2	p_2
0x0080	a_1	b_1	c_1	d_1	e_1	f_1	g_1	h_1	i_1	j_1	k_1	l_1	m_1	n_1	o_1	p_1
0x0040	a_0	b_0	c_0	d_0	e_0	f_0	g_0	h_0	i_0	j_0	k_0	l_0	m_0	n_0	o_0	p_0
0x0000	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Back to cache considerations

Memory view for container of objects

```
struct A { float a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p; }
std::vector<A> v;
```

Each `A` corresponds to a cache line (64 bytes, 16 floats)

0x01C0	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
0x0180	a_5	b_5	c_5	d_5	e_5	f_5	g_5	h_5	i_5	j_5	k_5	l_5	m_5	n_5	o_5	p_5
0x0140	a_4	b_4	c_4	d_4	e_4	f_4	g_4	h_4	i_4	j_4	k_4	l_4	m_4	n_4	o_4	p_4
0x0100	a_3	b_3	c_3	d_3	e_3	f_3	g_3	h_3	i_3	j_3	k_3	l_3	m_3	n_3	o_3	p_3
0x00C0	a_2	b_2	c_2	d_2	e_2	f_2	g_2	h_2	i_2	j_2	k_2	l_2	m_2	n_2	o_2	p_2
0x0080	a_1	b_1	c_1	d_1	e_1	f_1	g_1	h_1	i_1	j_1	k_1	l_1	m_1	n_1	o_1	p_1
0x0040	a_0	b_0	c_0	d_0	e_0	f_0	g_0	h_0	i_0	j_0	k_0	l_0	m_0	n_0	o_0	p_0
0x0000	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Computing $\sum g_n$ requires usage of all cache lines

Structure o Arrays (SoA) approach

Let's put together what goes together

```
struct As {
  std::vector<float> a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p;
} v;
```

Memory now looks like this :

0x01C0	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
0x0180	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}	e_{11}	e_{12}	-	-	-
0x0140	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	-	-	-
0x0100	g_0	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8	g_9	g_{10}	g_{11}	g_{12}	-	-	-
0x00C0	h_0	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}	h_{11}	h_{12}	-	-	-
0x0080	i_0	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9	i_{10}	i_{11}	i_{12}	-	-	-
0x0040	j_0	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}	j_{12}	-	-	-
0x0000	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Structure o Arrays (SoA) approach

Let's put together what goes together

```
struct As {
  std::vector<float> a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p;
} v;
```

Memory now looks like this :

0x01C0	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
0x0180	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}	e_{11}	e_{12}	-	-	-
0x0140	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	-	-	-
0x0100	g_0	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8	g_9	g_{10}	g_{11}	g_{12}	-	-	-
0x00C0	h_0	h_1	h_2	h_3	h_4	h_5	h_6	h_7	h_8	h_9	h_{10}	h_{11}	h_{12}	-	-	-
0x0080	i_0	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9	i_{10}	i_{11}	i_{12}	-	-	-
0x0040	j_0	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}	j_{12}	-	-	-
0x0000	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Computing $\sum g_n$ uses a single cache line

Consequences

- only one line loaded in L1 cache
- only one line dropped from that cache to fit the new one
- better chances to find data in cache for next instruction
- potential gain : factor 2 to 100 on memory access

Main lesson

- Colocate in memory what is used at the same time
- Drawback : optimization of the memory structure depends on the consumer

Detection of suboptimal allocations

The main tools : profilers

Find out where allocation really costs time

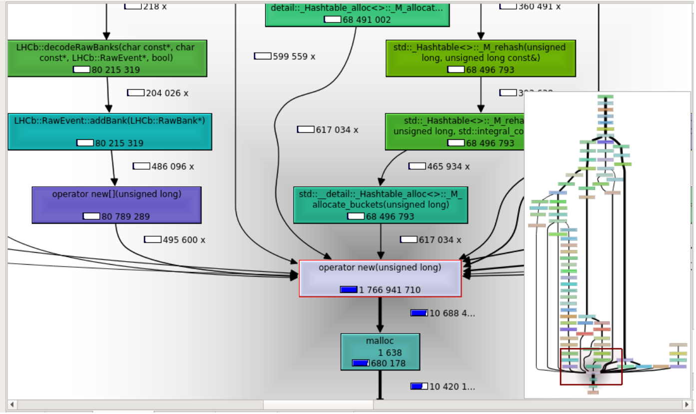
vtune from Intel

- uses internal processor counters to see where time is spent
- in particular number of cycles spent in memory allocations
- and cache misses

callgrind - open source

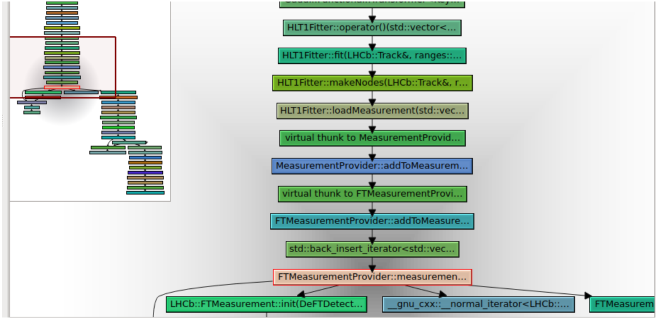
- simulates a processor and allows to count what is going on
- in particular number of cycles spent in memory allocations
- and (simulated) cache misses

callgrind in practice



- graph of function calls leading to memory allocation
- and the time spent for each case

callgrind in practice



- all comes from the HLT1Fitter, where you find :

```
unsigned int HLT1Fitter::fit(LHCb::Track& ...) const {
  // Store results of the Kalman fit
  std::vector<LHCb::Measurement*> measurements;
```

Conclusions

- Memory allocation/deallocations are not cheap
- Optimizing them can lead to substantial gains
- key directions are :
  - container of objects
  - preallocate containers aka "reservation"
  - optimize your data structures for caches, aka SoA
  - use profilers to detect potential issues