Optimizing existing large codebase

Sébastien Ponce
sebastien.ponce@cern.ch

CERN

Thematic CERN School of Computing 2025

Outline

1. Measuring Performance
   - What is performance ?
   - Tools available
   - Finding bottlenecks
2. Code modernization
3. Improving Memory Handling
   - Context
   - Containers and memory
   - Container reservation
   - Detecting offending code
4. The nightmare of thread safety
   - Context and constraints
   - Identifying problems
   - Solving problems
   - Thread contention
5. Low level optimizations
   - Scope and target
   - How to measure ?
   - Improving
6. Conclusion

Goal of this course

- make the theory explained so far more concrete
- and adapt it to the special case of
  - dealing with large projects
  - dealing with legacy code
- I'll only talk of C^{++} projects

Specificity of the exercise

- Dealing with large code base (Mloc)
  - most of them unknown to you
  - and (usually) not supported by anyone
- Dealing with old code
  - using old fashion coding style (e.g. FORTRAN like)
  - modified n times, grew organically
- Target latest hardware
  - many cores
  - hyperthreading / superscalar
  - vectorization

Overall strategy

First measure !

- understand where time is spent
- understand the main limitations

Then attack these limitations

- modernize the code
- optimizing memory handling
- optimizing parallelism
- optimizing low level code

Measuring Performance

1 Measuring Performance
- What is performance ?
- Tools available
- Finding bottlenecks

2 Code modernization

3 Improving Memory Handling

4 The nightmare of thread safety

5 Low level optimizations

6 Conclusion

Defining our performance

Key question is : what is performance

- simply going faster ?

Defining our performance

Key question is : what is performance

- simply going faster ?
  - not at all costs (money, physics results)

Defining our performance

Key question is : what is performance

- simply going faster ?
  - not at all costs (money, physics results)
- making better use of the hardware

Defining our performance

Key question is : what is performance

- simply going faster ?
  - not at all costs (money, physics results)
- making better use of the hardware
  - most of the time hardware is cheaper than people !

Defining our performance

Key question is : what is performance

- simply going faster ?
  - not at all costs (money, physics results)
- making better use of the hardware
  - most of the time hardware is cheaper than people !
- you need to define your "Key Performance Indicators"
  - e.g. nb Evts / s / $ with constant man power for a trigger
- and get a clear idea of your different costs
  - flops/$ of your machines
    - including network, cabling, cooling, buildings, ...
  - human costs
  - cost of transition
  - ...

Measuring our software

Many parameters can be measured

- overall timing
- memory usage and cache efficiency
- CPU efficiency (Cycles per instructions, vectorization level)
- level of parallelism, usage of the different cores
- I/O limitations if any

For each of them, you need

- both overall data and detailed split per code unit
- per item, per core and full machine measurement

How to measure

The counters approach

- use CPU counters to find out what happened during actual execution
- do not slow down execution, so only do sampling

The software instrumentation

- run your code in a "virtual" environment
- measure everything precisely
- at the cost of speed

Counters approach in practice

- give precise timing of a realistic execution on your CPU
  - using real cache prediction, actual vectorization, ...
  - using real CPU behavior (e.g. downclocking when overheating...)
- allows to measure CPI (**C**ycles **P**er **I**nstruction) and low level behavior in general (caching, pipelining)
- but data is only statistical
  - so you need sufficient statistics
  - also not always reproducible, so hard to compare
    - e.g. first test on cold processor, second on warm one
- Main tools available : perf and variants, Intel VTune

Software instrumentation in practice

- give precise measurements of where you spend instructions
  - including many details
  - reproducible, so your can compare stuff
- but not always realistic
  - no real timing, only instructions count
  - memory caching is only simulated, often far from real case
  - no clue on low level efficiency (CPI in particular)
  - and gives no clue on hardware / OS behavior
- Main tool available : valgrind family

Finding bottlenecks

Understand where we can improve

- analyze each part of the software
- in order to find out where most time is spent
- and understand whether it can be improved

Most usual bottlenecks

From biggest to lowest impact (usually)

- IO
- Memory
- Parallelization
- Low level behavior : vectorization, cache behavior, high CPI

Code modernization

1 Measuring Performance

2 Code modernization

3 Improving Memory Handling

4 The nightmare of thread safety

5 Low level optimizations

6 Conclusion

Make use of latest C++features

- C++has evolved dramatically between 2010 and now
- five new versions : C++11, C++14, C++17, C++20, C++23
- a LOT of new features targeting performance
  - move semantic
  - threading library
  - variadic templates
  - vectorization coming ?
- converting existing code may already bring speed
- see previous courses for technical details
- see HSF C++course if you're not at ease with the language

Cleanup your code

- While reviewing the code for converting to latest C++:
  - drop unused code
  - drop unnecessary code
    - e.g. do I really need to sort by hits here ?
  - drop too generic APIs if they are finally not needed
  - replace virtual inheritance with templating when possible
  - consider dropping use of unmaintained libraries
- It is very often surprising how much you gain there

Improving Memory Handling

1 Measuring Performance

2 Code modernization

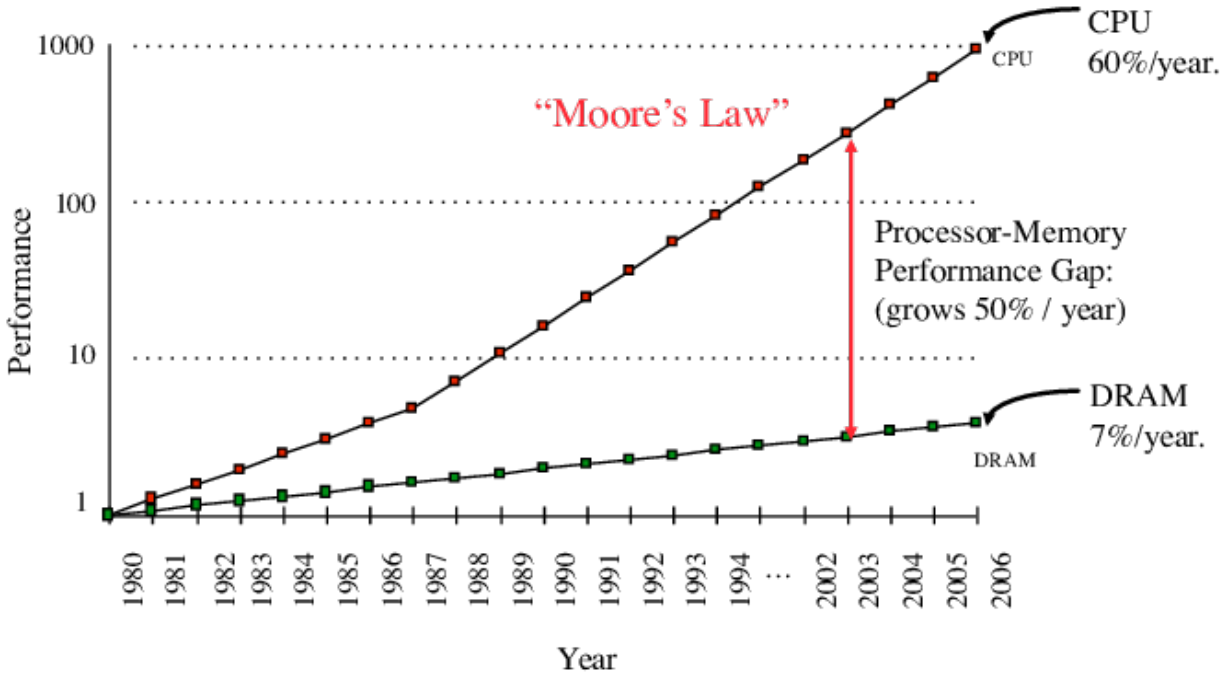
3 Improving Memory Handling
- Context
- Containers and memory
- Container reservation
- Detecting offending code

4 The nightmare of thread safety

5 Low level optimizations

6 Conclusion

Evolution of memory in the past decades

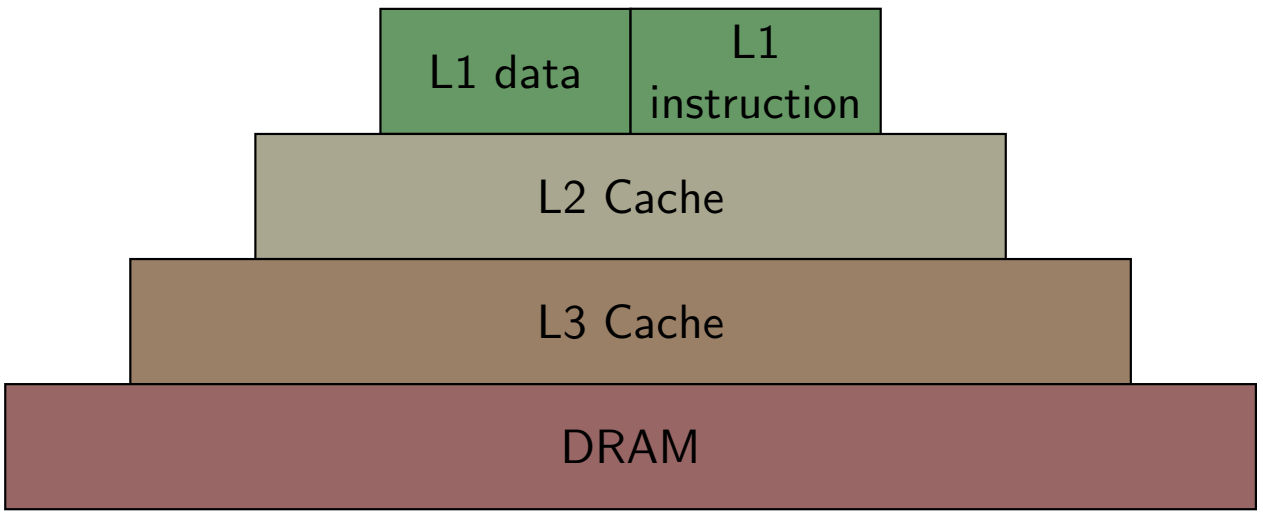


Due to Moore's law in the 80s and 90s, there is a gap between CPU and memory performances

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

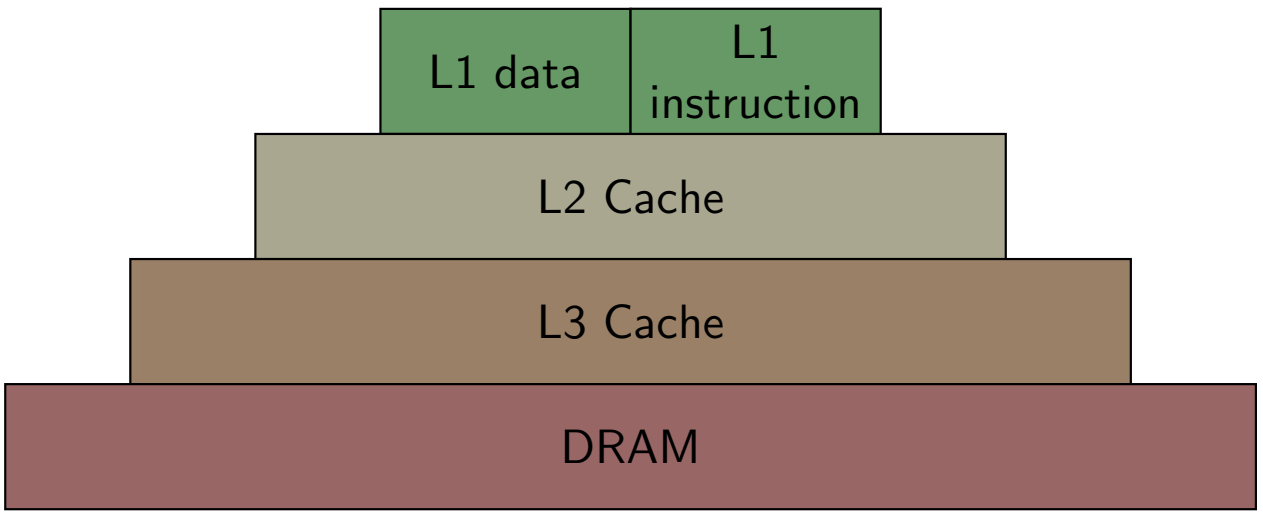
Typical cache structure



	size	latency
L1	64 kB	4 cycles
L2	256 kB	10 cycles
L3	10 MB	40 cycles
DRAM	64 GB	400 cycles

Typical data, on an Haswell architecture

Typical cache structure



	size	latency
L1 data / L1 instruction	64 kB	4 cycles
L2 Cache	256 kB	10 cycles
L3 Cache	10 MB	40 cycles
DRAM	64 GB	400 cycles

Typical data, on an Haswell architecture

Practical consequence in C++

Guidelines

- we want as few heap memory allocations as possible
  - stack usage is much better !
- we want continuous memory blocks, specially for containers
  - that means containers of objects, no pointers involved
  - e.g. `vector<Obj*>` or `array<vector<Obj>>` are banned !

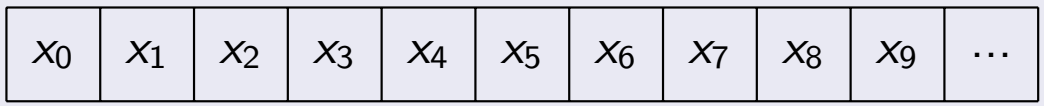
2 main rules

- use container of objects, not of pointers
  - use (const) references everywhere
  - avoid any unnecessary copy of data
    - including implicit ones
- use container reservation

Container of objects in memory

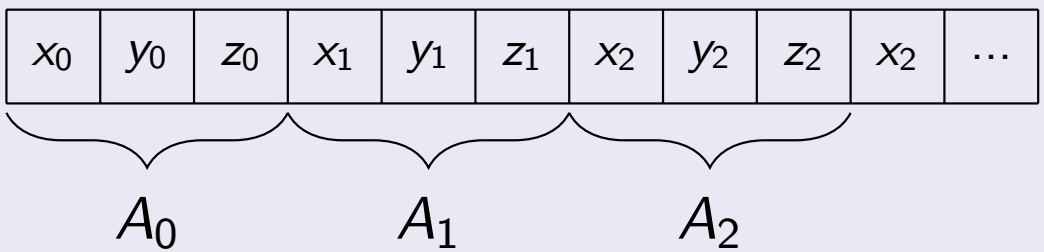
Simple vector case

```
std::vector<int> v;
```



Vector of objects

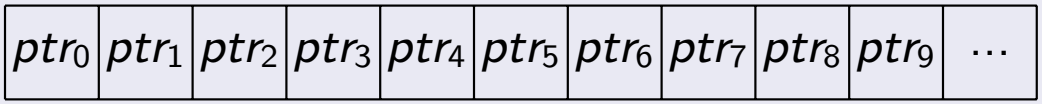
```
struct A { float x, y, z; };
std::vector<A> v;
```



Container of pointers in memory

Naïve view

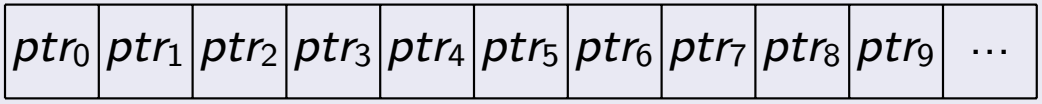
```
struct A { float x, y, z; };
std::vector<A*> v;
```



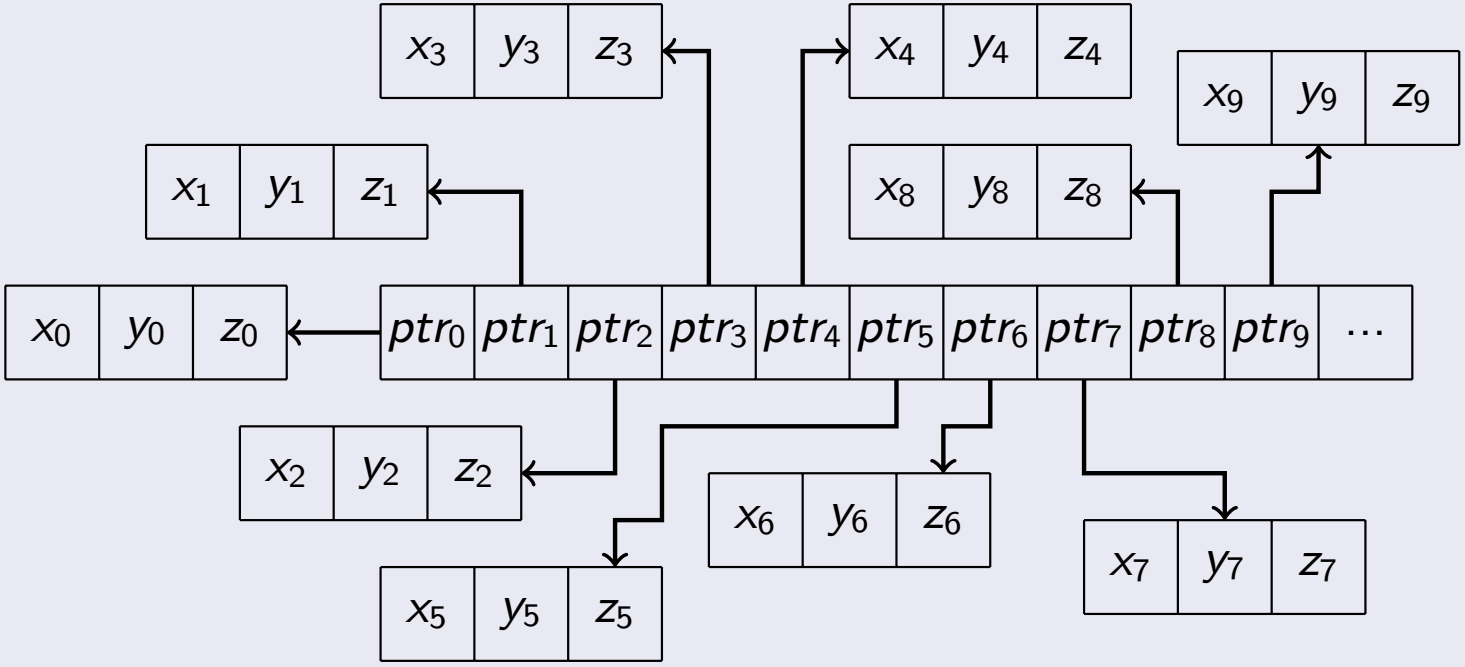
Container of pointers in memory

Naïve view

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



Realistic view



Container of objects in cache

Memory view for `vector<A>`

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	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·

All data are nicely collocated in cache

Container of pointers in cache

Memory view for `vector<A*>`

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																

Cache nightmare : data is completely sparse

Note from Andrzej : this is already optimistic

Container with no reservation

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

Container with no reservation

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

Construction

Default constructor creates and empty vector, with no storage

start	0x0
finish	0x0
end_of_storage	0x0

Container with no reservation

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

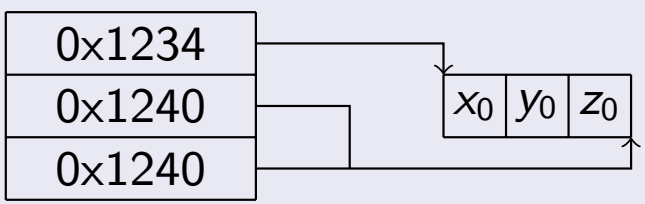
Construction

Default constructor creates and empty vector, with no storage

start	0x0
finish	0x0
end_of_storage	0x0

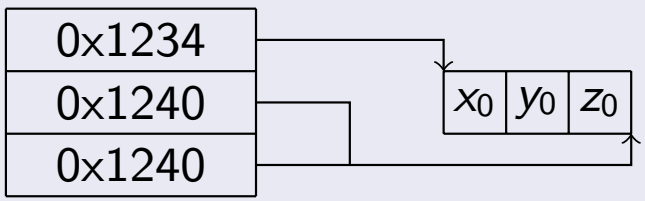
First push

allocates storage for the first element only !



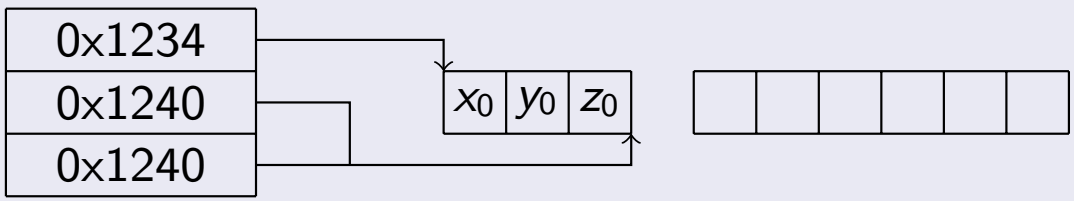
Container with no reservation

Second push



Container with no reservation

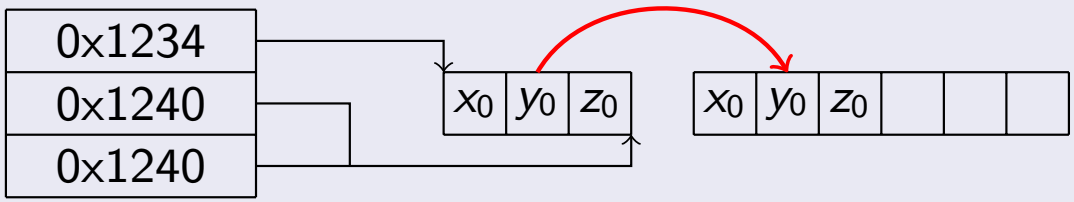
Second push



1. allocate new piece of memory for 2 items

Container with no reservation

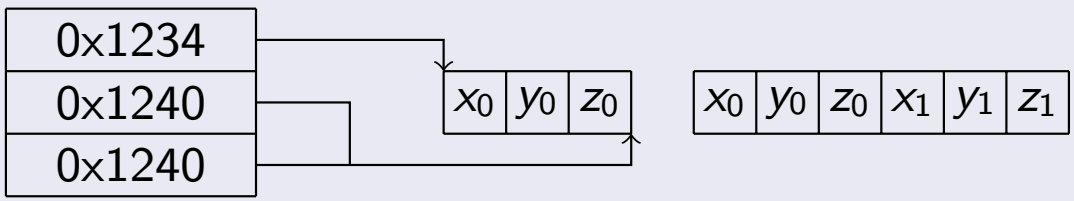
Second push



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

Container with no reservation

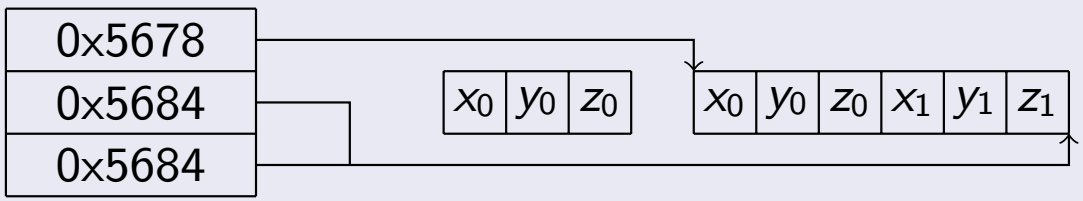
Second push



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

Container with no reservation

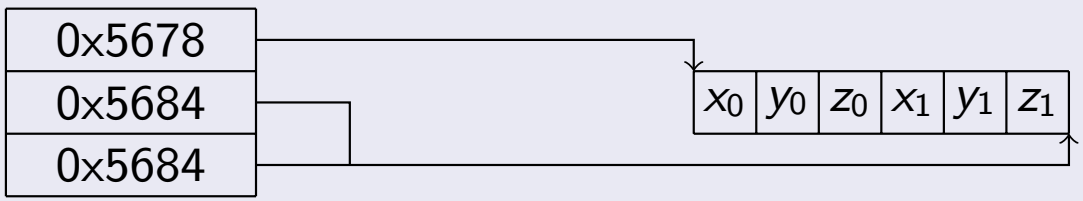
Second push



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

Container with no reservation

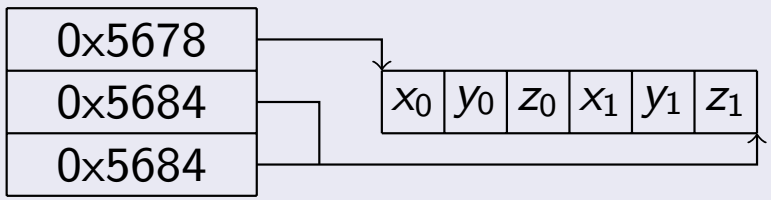
Second push



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

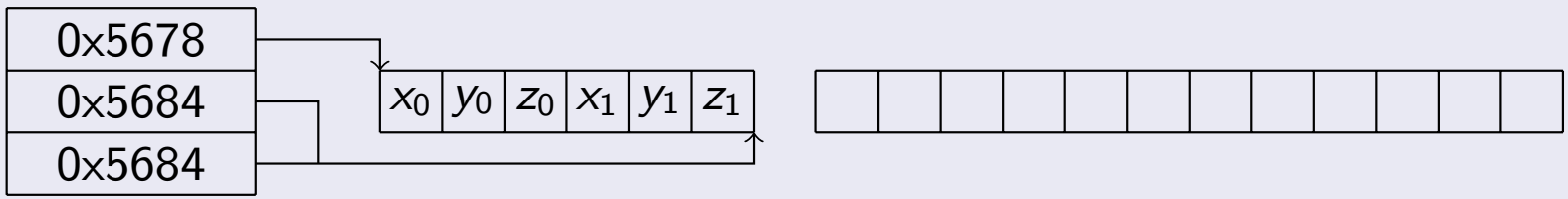
Container with no reservation

Third push



Container with no reservation

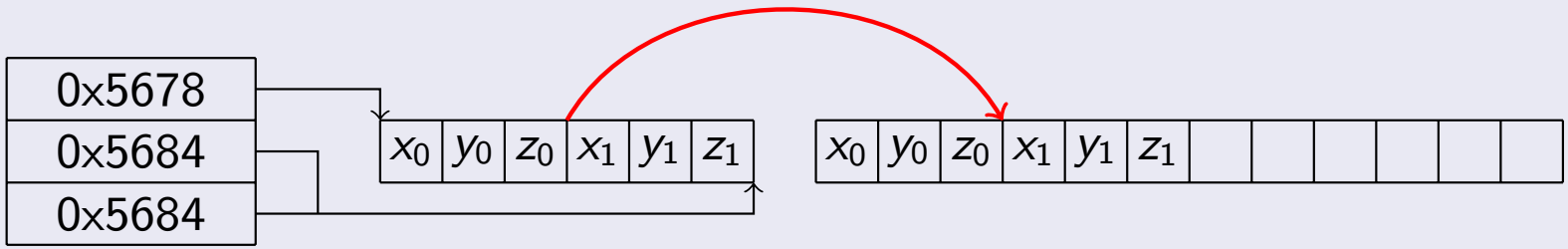
Third push



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

Container with no reservation

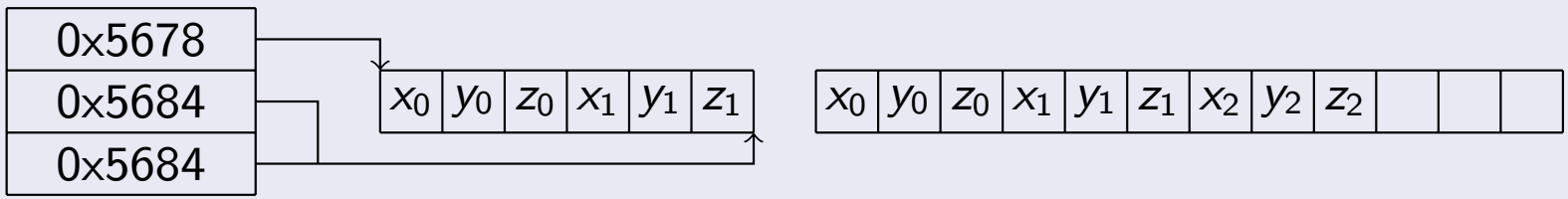
Third push



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

Container with no reservation

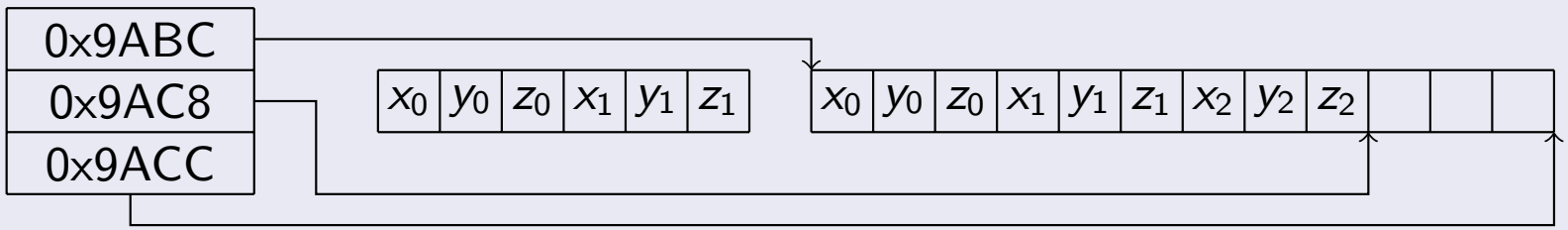
Third push



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

Container with no reservation

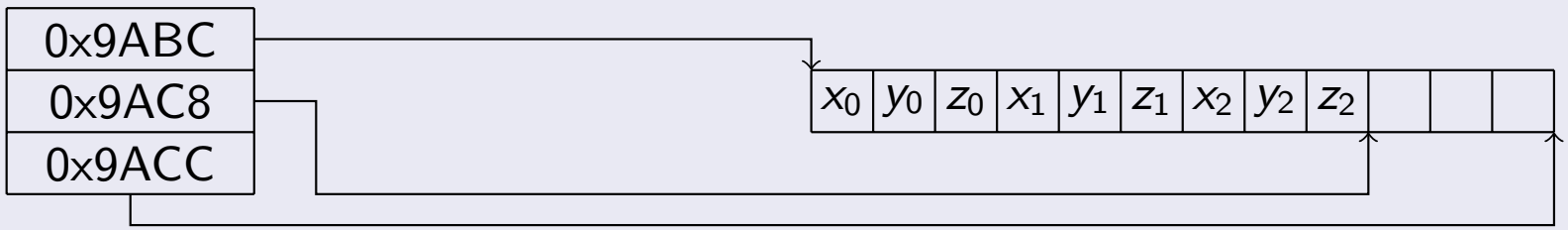
Third push



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

Container with no reservation

Third push



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

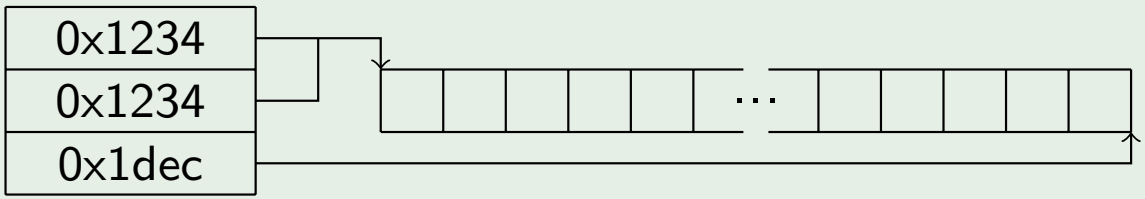
Container with proper reservation

Construction and reservation

- you can avoid all that thanks to reserve

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

- ensures single allocation, no copies, no reallocation



Without second line

- first item would have be copied 10 times !
- 1023 items would have been copied in total
- 11 pieces of memory allocated, 10 released

Avoiding copy when filling container

What should be avoided

```
1  std::vector<A> v;
2  v.reserve(10);
3  A tmp{args};
4  v.push_back(tmp);
```

Avoiding copy when filling container

What should be avoided

```
1  std::vector<A> v;
2  v.reserve(10);
3  A tmp{args};
4  v.push_back(tmp);
```

What actually happens :

- allocate space in the vector (line 2)
- allocate space for the temporary A object (line 3)
- call A constructor (line 3)
- call copy constructor for A (line 4)
- deallocate temporary A (end of scope)
- using `std::move(tmp)` on last line not necessarily better

Proper solution for vectors

In place construction

```
1  std::vector<A> v;
2  v.reserve(10);
3  v.emplace_back(args);
```

What actually happens :

- allocate space in the vector
- call constructor for A
  - using args as the constructor arguments
  - using the space allocated in the vector

For the record, this is using variadic templates, new in C++11

Proper solution for maps

In place construction + piecewise construct + forward_as_tuple

```
1  std::map<int,A> m;
2  m.emplace(piecewise_construct,
3            make_tuple(5),
4            forward_as_tuple(args));
```

- emplace_back now creates an std::pair
- piecewise_construct constructs the 2 items of the pair in place
- forward_as_tuple prevents a copy of args
  - by using a tuple of references

Detecting memory offending code

Look at your measurements !

- how much time do you spend in malloc/new/free/delete/... ?
  - more than a few % ? Room for improvement !
- what is your last level cache miss rate ?
  - 1 % or more ? Room for improvement !
- drill down to the method where the numbers are bad
- find the responsible container / data structure

Detecting memory offending code

Look at your measurements !

- how much time do you spend in malloc/new/free/delete/... ?
  - more than a few % ? Room for improvement !
- what is your last level cache miss rate ?
  - 1 % or more ? Room for improvement !
- drill down to the method where the numbers are bad
- find the responsible container / data structure

Function	Effective Time
operator new	16.7%
_int_free	8.3%
PrPixelTracking::bestHit	5.7%
PrForwardTool::collectAllXHits	5.7%
PVSeed3DTool::getSeeds	2.7%
PrStoreFTHit::storeHits	4.5%

The nightmare of thread safety

1 Measuring Performance

2 Code modernization

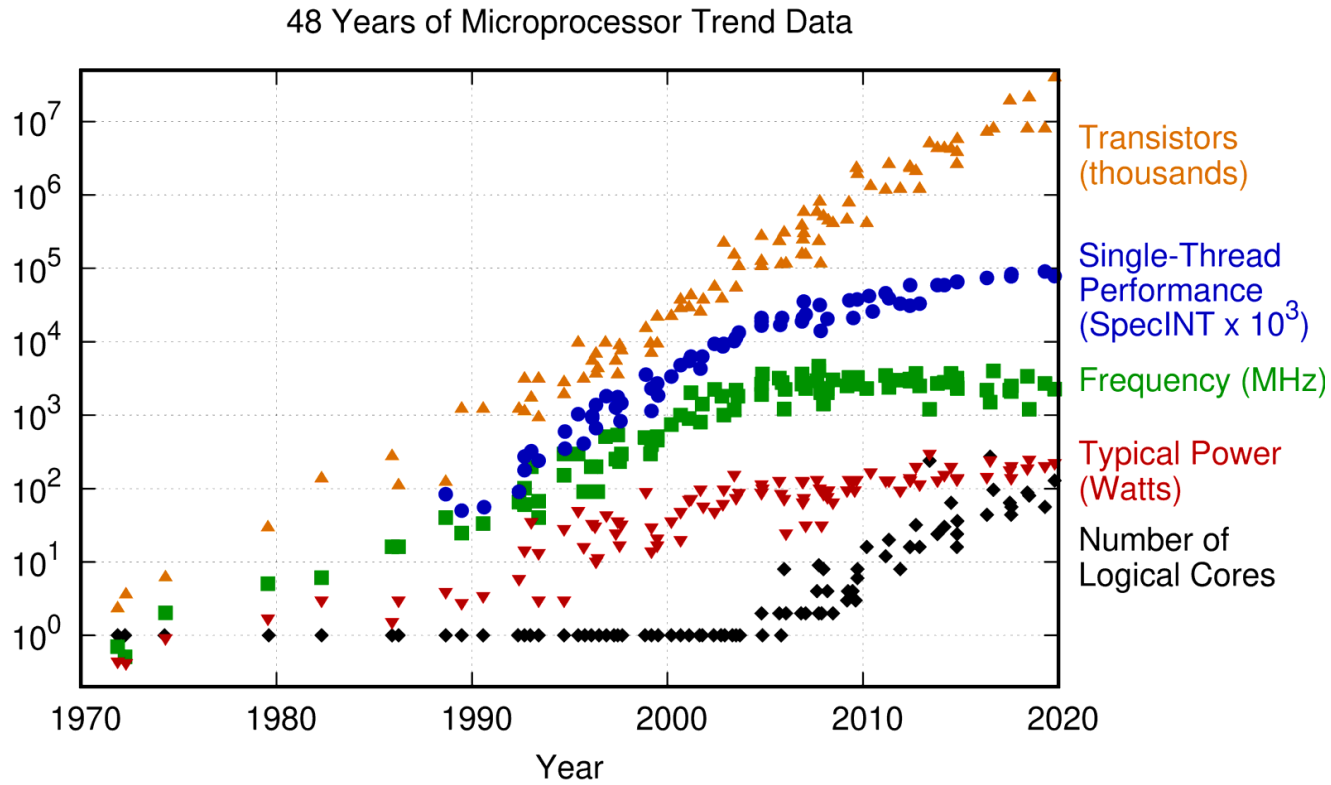
3 Improving Memory Handling

4 The nightmare of thread safety
- Context and constraints
- Identifying problems
- Solving problems
- Thread contention

5 Low level optimizations

6 Conclusion

Evolution of CPUs in the past decades



source : `https://github.com/karlrupp/microprocessor-trend-data`

- clock speed is now fixed
- flops come now from parallel processing
- and especially from many cores

Adapting existing software : multiprocessing

Cheap and easy approach

- works when you can split your data
- just process the pieces in parallel
- launching your original application n times
  - where n is roughly the number of cores
- and glue the results together

Main problem

- the memory usage (amount and cache efficiency)

Adapting existing software : multithreading

Less easy but rewarding approach

- remember threads share their heap
- your memory is mainly "read only" during processing
  - all code and libraries
  - detector geometry, conditions
- threads share this common memory
  - 40 threads instead of 40 jobs means memory usage divided by 40 !

Main problem

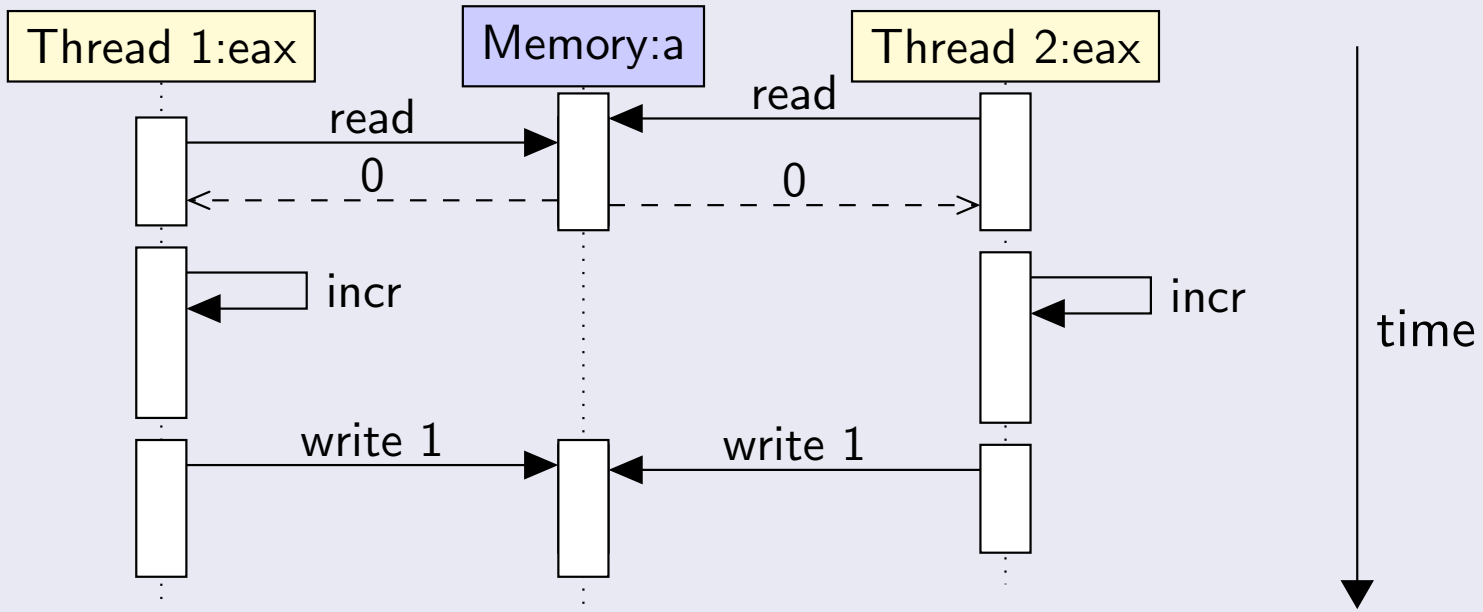
- race conditions on non read-only memory parts

Timing of ++ operation

Code

```
1  eax = a          // memory to register copy
2  increase eax     // increase (atomic CPU instruction)
3  a = eax          // copy back to memory
```

For 2 threads



What is not thread/safe ?

Everything ! unless explicitely stated

Non thread-safe code

- heap
- STL constainers
  - they use the heap !
  - some functions may be thread safe though
- global (non const) variables
- any member of a C^{++}class modified in a non const method
  - the method may be called concurrently by two threads
- `constcast`, singletons, caches, ...
- many C library calls (e.g. strtok, ctime, ...)
- external libearies
  - random number generators (most of them)

Converting to multithreading

Identify all shared state

and secure them

Identifying problematic shared states

A very hard topic !

Identifying problematic shared states

A very hard topic !

Level 0 : use gdb and debug crashes

- but race conditions do not happen at every run
- and crashes is the lucky case, wrong results is the other

Identifying problematic shared states

A very hard topic !

Level 0 : use gdb and debug crashes

- but race conditions do not happen at every run
- and crashes is the lucky case, wrong results is the other

Level 0.5 : helgrind

- detects race conditions that did not happen
- but only in code that ran (coverage issue)
- and is very slow

Identifying problematic shared states

Actual (partial) solution : use the language

- C++11 constness is different from original one
- it now means "visibly const and race condition free"
- that is the "visible" state is not modified
- and internally thread safety is guaranteed if any thing changes

Identifying problematic shared states

Actual (partial) solution : use the language

- C++11 constness is different from original one
- it now means "visibly const and race condition free"
- that is the "visible" state is not modified
- and internally thread safety is guaranteed if any thing changes

Bottom line : proper use of constness can save us !

- especially constness of member functions
- it ensures they are reentrant
- it will tell us at compile time if not :-)

Practically identifying shared state

- introduce constness everywhere
- look at compiler errors

Practically identifying shared state

- introduce constness everywhere
- look at compiler errors
- and look for globals, const does not apply here

Practically identifying shared state

- introduce constness everywhere
- look at compiler errors

- and look for globals, const does not apply here
- and do not forget special cases where constness is not respected...
  - mutables
  - const_cast, C casts

Practically identifying shared state

- introduce constness everywhere
- look at compiler errors

- and look for globals, const does not apply here
- and do not forget special cases where constness is not respected...
  - mutables
  - const_cast, C casts
- better be fluent with grep...

Typical legacy code

```
1  /// hack to allow for tools with non-const interfaces
2  template <typename IFace>
3  IFace* fixup(const ToolHandle<IFace>& iface) {
4    return &const_cast<IFace&>(*iface);
5  }
6
7  void CaloHypoNtp::operator()(...) const {
8    ...
9    if( m_checker )
10     fixup(m_2MC)->from(hypo)->descriptor();
11   ...
12 }
```

Securing a shared state

non optimal way : locks and mutexes

- serializes access to given piece of code
- fine under low pressure
- but costly and leads to contention under high pressure
  - see later

Better way

- analyze the data race, extract the state
- can it be replicated/moved to higher level ?

State replication option

When to use it

- when the state can be replicated
- so not used as a synchronization point
- for heavily used, small state

Example : random number generator

- state is small, does not need to be shared
- so you can have one generator per thread
- or modify the random generator
  - and use thread local storage for the state
  - although may be more costful

Moving state to higher level

When to use it

- when the state can be replicated
- so not used as a synchronization point
- fine with small and large states
- typically for caches, when the state is data reused between calls

Principle

- simply make the state visible to the caller
- and initialize it one level higher on the stack
- so that you have local states per thread

Think of the state as hidden data passed magically to your calls, just make it explicit

State moving option example

Original class

```
1  class CachedAccessor {
2    mutable CachedObject m_myCache;
3    Item get(const KeyObject& key) const {
4      ...
5      m_myCache.update(...);
6      ...
7    }
8  };
```

Usage

```
1  CachedAccessor accessor{dataSource};
2  for(const auto& key: keyList) {
3    auto item = accessor.get(key);
4    ...
5  }
```

State extraction to higher level

Thread safe version of the class

```
1  class CachedAccessor {
2    Item get (const KeyObject& key, CachedObject& cache) const {
3      cache.update(... key ...);
4      ...
5    }
6  };
```

thread safe usage

```
1  CachedAccessor accessor{dataSource};
2  CachedObject cache;
3  for(const auto& key: keyList) {
4    auto item = accessor.get(key, cache);
5    ...
6  }
```

What do we call contention ?

- a situation where all hardware resources cannot be used efficiently
- because of one bottleneck in the chain
  - called point of contention
- can be anything
  - hardware : network, I/O, ...
  - lines of code, typically with a lock
  - architectural : one producer not fast enough blocking many consumers
- actually can be anything shared between threads

How to detect contention ?

- benchmark the code
- look at scalability
- look at thread usage

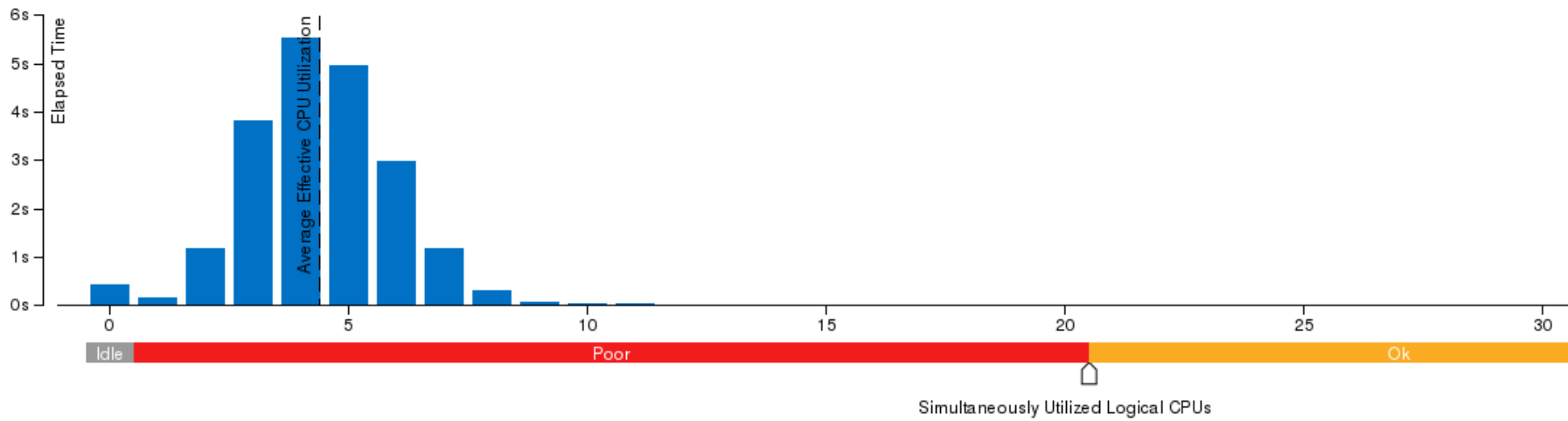
How to detect contention ?

- benchmark the code
- look at scalability
- look at thread usage

Example output of 40 threads on 20 physical cores' system

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



Where is the contention ?

Much more difficult question !

- you need to find the hotspot
- without any clue of its type
- look for
  - high CPU in a single thread
  - lengths of your queues (too full, too empty ?)
  - global state of your machine (I/O, network)

Example of memory based contention

Figure: Table of the main CPU consumers in the application

Function	Effective Time
operator new	16.7%
_int_free	8.3%
PrPixelTracking::bestHit	5.7%
PrForwardTool::collectAllXHits	5.7%
PVSeed3DTool::getSeeds	2.7%
PrStoreFTHit::storeHits	4.5%

Example of memory based contention

Figure: Table of the main CPU consumers in the application

Function	Effective Time
operator new	16.7%
_int_free	8.3%
PrPixelTracking::bestHit	5.7%
PrForwardTool::collectAllXHits	5.7%
PVSeed3DTool::getSeeds	2.7%
PrStoreFTHit::storeHits	4.5%

What was happening :

- too many small memory allocations across threads
- new has an internal locking for thread safety
- and cannot cope !

Example of memory based contention

Figure: Table of the main CPU consumers in the application

Function	Effective Time
operator new	16.7%
_int_free	8.3%
PrPixelTracking::bestHit	5.7%
PrForwardTool::collectAllXHits	5.7%
PVSeed3DTool::getSeeds	2.7%
PrStoreFTHit::storeHits	4.5%

What was happening :

- too many small memory allocations across threads
- new has an internal locking for thread safety
- and cannot cope !

Another good reason to care about memory allocation !

Low level optimizations

1 Measuring Performance

2 Code modernization

3 Improving Memory Handling

4 The nightmare of thread safety

5 Low level optimizations
- Scope and target
- How to measure ?
- Improving

6 Conclusion

Scope and target

- improve usage of the hardware for a given code
- that includes in particular
  - better caching of memory accesses
  - better usage of pipelining and superscalar features
  - better vectorization
- in principle, the compiler does all that for you
- but you have better high level knowledge of the code !

Measuring low level efficiency

CPI

- **C**ycles **P**er **I**nstruction
- should be smaller when pipelining and superscalar features are used
- but typical values highly depend on the type of code

Cache misses

- should be really low (<1%)
- Andrzej used to say : "5% is catastrophic"

Vectorization

- look at specific counters in vtune/valgrind (extension)
- check the assembly code directly looking for SIMD instruction codes

How to improve

By acting mainly on 2 constraints that the compiler has to obey

your data dependencies

- you need close-by operations to be independent
- compiler code reordering is limited
- and pipelining/superscalar need that

your data structure

- especially true for caching and vectorization
- your data structures should group together what goes together
- Called SoA (**S**tructure **o**f **A**rrays) approach

In both cases : work more in parallel !

Conclusion

1 Measuring Performance
2 Code modernization
3 Improving Memory Handling
4 The nightmare of thread safety
5 Low level optimizations
6 Conclusion

Conclusion

Key messages of the day

- typical large old code can benefit a lot of optimizations
  - typically speedup of > 2 !
- main lines of optimizations is reworking the data structures
  - for optimizing memory allocation
  - for allowing more parallelism/vectorization
- handling multicores typically goes through threading
  - huge effort to be made to ensure thread safety
  - constness can greatly help