Optimizing existing large codebase

Sébastien Ponce
sebastien.ponce@cern.ch

CERN

Thematic CERN School of Computing 2019
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 by Danilo and Andrzej 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
  - ...
Optimizing existing large codebase

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 (Cycles Per Instruction) 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
Optimizing existing large codebase

Code modernization

1. Measuring Performance
2. Code modernization
3. Improving Memory Handling
4. The nightmare of thread safety
5. Low level optimizations
6. Conclusion
C++ has evolved dramatically between 2010 and now
three new versions: C++11, C++14, C++17
a LOT of new features targeting performance
  - move semantic
  - threading library
  - variadic templates
  - vectorization coming!
converting existing code may already bring speed
see Danilo’s course for technical details
see my extended C++ course if you’re not at ease with the language
Cleanup your code

While reviewing the code for converting to C++17:
- 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
Optimizing existing large codebase

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

- **L1 data**
- **L1 instruction**
- **L2 Cache**
- **L3 Cache**
- **DRAM**

<table>
<thead>
<tr>
<th>Size</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 kB</td>
<td>4 cycles</td>
</tr>
<tr>
<td>256 kB</td>
<td>10 cycles</td>
</tr>
<tr>
<td>10 MB</td>
<td>40 cycles</td>
</tr>
<tr>
<td>64 GB</td>
<td>400 cycles</td>
</tr>
</tbody>
</table>

Typical data, on an Haswell architecture

Optimizing existing large codebase
Typical cache structure

L1 data  | L1 instruction
---------|---------------
L2 Cache
L3 Cache
DRAM

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

Typical data, on an Haswell architecture
Optimizing existing large codebase

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

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

| x₀ | x₁ | x₂ | x₃ | x₄ | x₅ | x₆ | x₇ | x₈ | x₉ | ... |

Vector of objects

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

| x₀ | y₀ | z₀ | x₁ | y₁ | z₁ | x₂ | y₂ | z₂ | x₃ | ... |

A₀  A₁  A₂
Container of pointers in memory

Naïve view

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

```plaintext
ptr0  ptr1  ptr2  ptr3  ptr4  ptr5  ptr6  ptr7  ptr8  ptr9  ...
```
Naïve view

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

Realistic view

```
ptr0  ptr1  ptr2  ptr3  ptr4  ptr5  ptr6  ptr7  ptr8  ptr9  ...
```

```
x_0  y_0  z_0
ptr0  ptr1  ptr2  ptr3  ptr4  ptr5  ptr6  ptr7  ptr8  ptr9  ...
```

```
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  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
```
Container of objects in cache

Memory view for `vector<A>`

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000</td>
<td></td>
</tr>
<tr>
<td>0x00040</td>
<td></td>
</tr>
<tr>
<td>0x00080</td>
<td></td>
</tr>
<tr>
<td>0x00C0</td>
<td></td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>Address</th>
<th>Data Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>0x0008</td>
<td></td>
</tr>
<tr>
<td>0x0010</td>
<td></td>
</tr>
<tr>
<td>0x0014</td>
<td></td>
</tr>
<tr>
<td>0x0018</td>
<td></td>
</tr>
<tr>
<td>0x0020</td>
<td></td>
</tr>
<tr>
<td>0x0024</td>
<td></td>
</tr>
<tr>
<td>0x0040</td>
<td></td>
</tr>
<tr>
<td>0x0044</td>
<td></td>
</tr>
<tr>
<td>0x0048</td>
<td></td>
</tr>
<tr>
<td>0x0050</td>
<td></td>
</tr>
<tr>
<td>0x0054</td>
<td></td>
</tr>
<tr>
<td>0x0058</td>
<td></td>
</tr>
<tr>
<td>0x0060</td>
<td></td>
</tr>
<tr>
<td>0x0064</td>
<td></td>
</tr>
<tr>
<td>0x0068</td>
<td></td>
</tr>
<tr>
<td>0x0070</td>
<td></td>
</tr>
<tr>
<td>0x0074</td>
<td></td>
</tr>
<tr>
<td>0x0078</td>
<td></td>
</tr>
<tr>
<td>0x0080</td>
<td></td>
</tr>
<tr>
<td>0x0084</td>
<td></td>
</tr>
<tr>
<td>0x0088</td>
<td></td>
</tr>
<tr>
<td>0x0090</td>
<td></td>
</tr>
<tr>
<td>0x0094</td>
<td></td>
</tr>
<tr>
<td>0x0098</td>
<td></td>
</tr>
<tr>
<td>0x00A0</td>
<td></td>
</tr>
<tr>
<td>0x00A4</td>
<td></td>
</tr>
<tr>
<td>0x00A8</td>
<td></td>
</tr>
<tr>
<td>0x00B0</td>
<td></td>
</tr>
<tr>
<td>0x00B4</td>
<td></td>
</tr>
<tr>
<td>0x00B8</td>
<td></td>
</tr>
<tr>
<td>0x00C0</td>
<td></td>
</tr>
<tr>
<td>0x00C4</td>
<td></td>
</tr>
<tr>
<td>0x00C8</td>
<td></td>
</tr>
<tr>
<td>0x00D0</td>
<td></td>
</tr>
<tr>
<td>0x00D4</td>
<td></td>
</tr>
<tr>
<td>0x00D8</td>
<td></td>
</tr>
<tr>
<td>0x00E0</td>
<td></td>
</tr>
<tr>
<td>0x00E4</td>
<td></td>
</tr>
<tr>
<td>0x00E8</td>
<td></td>
</tr>
<tr>
<td>0x00F0</td>
<td></td>
</tr>
<tr>
<td>0x00F4</td>
<td></td>
</tr>
<tr>
<td>0x00F8</td>
<td></td>
</tr>
<tr>
<td>0x0100</td>
<td></td>
</tr>
<tr>
<td>0x0104</td>
<td></td>
</tr>
<tr>
<td>0x0108</td>
<td></td>
</tr>
<tr>
<td>0x0110</td>
<td></td>
</tr>
<tr>
<td>0x0114</td>
<td></td>
</tr>
<tr>
<td>0x0118</td>
<td></td>
</tr>
<tr>
<td>0x0120</td>
<td></td>
</tr>
<tr>
<td>0x0124</td>
<td></td>
</tr>
<tr>
<td>0x0128</td>
<td></td>
</tr>
<tr>
<td>0x0130</td>
<td></td>
</tr>
<tr>
<td>0x0134</td>
<td></td>
</tr>
<tr>
<td>0x0138</td>
<td></td>
</tr>
<tr>
<td>0x0140</td>
<td></td>
</tr>
<tr>
<td>0x0144</td>
<td></td>
</tr>
<tr>
<td>0x0148</td>
<td></td>
</tr>
<tr>
<td>0x0150</td>
<td></td>
</tr>
<tr>
<td>0x0154</td>
<td></td>
</tr>
<tr>
<td>0x0158</td>
<td></td>
</tr>
<tr>
<td>0x0160</td>
<td></td>
</tr>
<tr>
<td>0x0164</td>
<td></td>
</tr>
<tr>
<td>0x0168</td>
<td></td>
</tr>
<tr>
<td>0x0170</td>
<td></td>
</tr>
<tr>
<td>0x0174</td>
<td></td>
</tr>
<tr>
<td>0x0178</td>
<td></td>
</tr>
<tr>
<td>0x0180</td>
<td></td>
</tr>
<tr>
<td>0x0184</td>
<td></td>
</tr>
<tr>
<td>0x0188</td>
<td></td>
</tr>
<tr>
<td>0x0190</td>
<td></td>
</tr>
<tr>
<td>0x0194</td>
<td></td>
</tr>
<tr>
<td>0x0198</td>
<td></td>
</tr>
<tr>
<td>0x01A0</td>
<td></td>
</tr>
<tr>
<td>0x01A4</td>
<td></td>
</tr>
<tr>
<td>0x01A8</td>
<td></td>
</tr>
<tr>
<td>0x01B0</td>
<td></td>
</tr>
<tr>
<td>0x01B4</td>
<td></td>
</tr>
<tr>
<td>0x01B8</td>
<td></td>
</tr>
<tr>
<td>0x01C0</td>
<td></td>
</tr>
<tr>
<td>0x01C4</td>
<td></td>
</tr>
<tr>
<td>0x01C8</td>
<td></td>
</tr>
<tr>
<td>0x01D0</td>
<td></td>
</tr>
<tr>
<td>0x01D4</td>
<td></td>
</tr>
<tr>
<td>0x01D8</td>
<td></td>
</tr>
<tr>
<td>0x01E0</td>
<td></td>
</tr>
<tr>
<td>0x01E4</td>
<td></td>
</tr>
<tr>
<td>0x01E8</td>
<td></td>
</tr>
<tr>
<td>0x01F0</td>
<td></td>
</tr>
<tr>
<td>0x01F4</td>
<td></td>
</tr>
<tr>
<td>0x01F8</td>
<td></td>
</tr>
<tr>
<td>0x0200</td>
<td></td>
</tr>
<tr>
<td>0x0204</td>
<td></td>
</tr>
<tr>
<td>0x0208</td>
<td></td>
</tr>
<tr>
<td>0x0210</td>
<td></td>
</tr>
<tr>
<td>0x0214</td>
<td></td>
</tr>
<tr>
<td>0x0218</td>
<td></td>
</tr>
<tr>
<td>0x0220</td>
<td></td>
</tr>
<tr>
<td>0x0224</td>
<td></td>
</tr>
<tr>
<td>0x0228</td>
<td></td>
</tr>
<tr>
<td>0x0230</td>
<td></td>
</tr>
<tr>
<td>0x0234</td>
<td></td>
</tr>
<tr>
<td>0x0238</td>
<td></td>
</tr>
<tr>
<td>0x0240</td>
<td></td>
</tr>
<tr>
<td>0x0244</td>
<td></td>
</tr>
<tr>
<td>0x0248</td>
<td></td>
</tr>
<tr>
<td>0x0250</td>
<td></td>
</tr>
<tr>
<td>0x0254</td>
<td></td>
</tr>
<tr>
<td>0x0258</td>
<td></td>
</tr>
<tr>
<td>0x0260</td>
<td></td>
</tr>
<tr>
<td>0x0264</td>
<td></td>
</tr>
<tr>
<td>0x0268</td>
<td></td>
</tr>
<tr>
<td>0x0270</td>
<td></td>
</tr>
<tr>
<td>0x0274</td>
<td></td>
</tr>
<tr>
<td>0x0278</td>
<td></td>
</tr>
<tr>
<td>0x0280</td>
<td></td>
</tr>
<tr>
<td>0x0284</td>
<td></td>
</tr>
<tr>
<td>0x0288</td>
<td></td>
</tr>
<tr>
<td>0x0290</td>
<td></td>
</tr>
<tr>
<td>0x0294</td>
<td></td>
</tr>
<tr>
<td>0x0298</td>
<td></td>
</tr>
<tr>
<td>0x02A0</td>
<td></td>
</tr>
<tr>
<td>0x02A4</td>
<td></td>
</tr>
<tr>
<td>0x02A8</td>
<td></td>
</tr>
<tr>
<td>0x02B0</td>
<td></td>
</tr>
<tr>
<td>0x02B4</td>
<td></td>
</tr>
<tr>
<td>0x02B8</td>
<td></td>
</tr>
<tr>
<td>0x02C0</td>
<td></td>
</tr>
<tr>
<td>0x02C4</td>
<td></td>
</tr>
<tr>
<td>0x02C8</td>
<td></td>
</tr>
<tr>
<td>0x02D0</td>
<td></td>
</tr>
<tr>
<td>0x02D4</td>
<td></td>
</tr>
<tr>
<td>0x02D8</td>
<td></td>
</tr>
<tr>
<td>0x02E0</td>
<td></td>
</tr>
<tr>
<td>0x02E4</td>
<td></td>
</tr>
<tr>
<td>0x02E8</td>
<td></td>
</tr>
<tr>
<td>0x02F0</td>
<td></td>
</tr>
<tr>
<td>0x02F4</td>
<td></td>
</tr>
<tr>
<td>0x02F8</td>
<td></td>
</tr>
</tbody>
</table>

Cache nightmare: data is completely sparse

Note from Andrzej: this is already optimistic
Container with no reservation

```cpp
struct A { float x, y, z; }
std::vector<A> v;
```
Container with no reservation

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

Construction

Default constructor creates and empty vector, with no storage

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>0x0</td>
</tr>
<tr>
<td>finish</td>
<td>0x0</td>
</tr>
<tr>
<td>end_of_storage</td>
<td>0x0</td>
</tr>
</tbody>
</table>
Container with no reservation

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

**Construction**

Default constructor creates and empty vector, with no storage

<table>
<thead>
<tr>
<th>start</th>
<th>0x0</th>
</tr>
</thead>
<tbody>
<tr>
<td>finish</td>
<td>0x0</td>
</tr>
<tr>
<td>end_of_storage</td>
<td>0x0</td>
</tr>
</tbody>
</table>

**First push**

allocates storage for the first element only!

```
0x1234 0x1240 0x1240
x0 y0 z0
```
Container with no reservation

Second push

```
0x1234
0x1240
0x1240
```

x0 y0 z0

allocate new piece of memory for 2 items

1

2

copy existing content

3

write new content

4

update pointers

5

Deallocate original piece of memory
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
Optimizing existing large codebase

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

<table>
<thead>
<tr>
<th>0x5678</th>
<th>x0</th>
<th>y0</th>
<th>z0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5684</td>
<td>x1</td>
<td>y1</td>
<td>z1</td>
</tr>
</tbody>
</table>

Container with no reservation

Third push

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

Allocate new piece of memory for 4 items:

- Allocate a new piece of memory for 4 items.
- Double the 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
Optimizing existing large codebase

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
Optimizing existing large codebase

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
  ```cpp
  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

```cpp
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 the last line is not necessarily better.
Avoiding copy when filling container

What should be avoided

```cpp
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. \texttt{std::map<int,A> m;}
2. \texttt{m.emplace(piecewise_construct,}
3. \texttt{ \texttt{make_tuple(5),}}
4. \texttt{ \texttt{forward_as_tuple(args));}}

- \texttt{emplace_back} now creates an \texttt{std::pair}
- \texttt{piecewise_construct} constructs the 2 items of the pair in place
- \texttt{forward_as_tuple} prevents a copy of \texttt{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

<table>
<thead>
<tr>
<th>Function</th>
<th>Effective Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator new</td>
<td>16.7%</td>
</tr>
<tr>
<td>_int_free</td>
<td>8.3%</td>
</tr>
<tr>
<td>PrPixelTracking::bestHit</td>
<td>5.7%</td>
</tr>
<tr>
<td>PrForwardTool::collectAllXHits</td>
<td>5.7%</td>
</tr>
<tr>
<td>PVSeed3DTool::getSeeds</td>
<td>2.7%</td>
</tr>
<tr>
<td>PrStoreFTHit::storeHits</td>
<td>4.5%</td>
</tr>
</tbody>
</table>
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

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

Source: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data
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
To Be Precise: Data Race

Standard language rules, §1.10/4 and /21:

- Two expression evaluations conflict if one of them modifies a memory location (1.7) and the other one accesses or modifies the same memory location.

- The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behaviour.
Simple Example

Concurrency can compromise correctness

- Two threads: A and B, a variable X (44)
- A adds 10 to a variable X
- B subtracts 12 to a variable X

![Diagram showing the desired sequence of operations and the resulting values for each thread combination.](image-url)
What is not Thread Safe?

Everything, unless explicitly stated!

In four words: Shared State Among Threads

Examples:

- Static non const variables

- **STL containers**
  - Some operations are thread safe, but useful to assume none is!
  - Very well documented (e.g. [http://www.cplusplus.com/reference](http://www.cplusplus.com/reference))

- Many random number generators (the stateful ones)

- Calls like: strtok, strerror, asctime, gmtime, ctime ...

- Some math libraries (statics used as cache for speed in serial execution…)

- Const casts, singletons with state: indication of unsafe policies

It sounds depressing. But there are several ways to protect thread unsafe resources!
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 anything 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 anything changes

Bottom line: proper use of constness can save us!

- especially constness of member functions
- it ensures they are reentrant
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

```cpp
    /// hack to allow for tools with non-const interfaces
    template <typename IFace>
    IFace* fixup(const ToolHandle<IFace>& iface) {
        return &const_cast<IFace&>(*iface);
    }

    void CaloHypoNtp::operator()(...) const {
        ...
        if( m_checker )
            fixup(m_2MC)->from(hypo)->descriptor();
        ...
    }
```
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
Optimizing existing large codebase

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**

```cpp
class CachedAccessor {
    mutable CachedObject m_myCache;
    Item get(const KeyObject& key) const {
        ...
        m_myCache.update(...);
        ...
    }
};
```

**Usage**

```cpp
CachedAccessor accessor{dataSource};
for(const auto& key: keyList) {
    auto item = accessor.get(key);
    ...
}
```
State extraction to higher level

Thread safe version of the class

class CachedAccessor {
    Item get (const KeyObject& key, CachedObject& cache) const {
        cache.update(... key ...);
        ...
    }
};

thread safe usage

CachedAccessor accessor{dataSource};
CachedObject cache;
for(const auto& key: keyList) {
    auto item = accessor.get(key, cache);
    ...
}

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 the 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Effective Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator new</td>
<td>16.7%</td>
</tr>
<tr>
<td>_int_free</td>
<td>8.3%</td>
</tr>
<tr>
<td>PrPixelTracking::bestHit</td>
<td>5.7%</td>
</tr>
<tr>
<td>PrForwardTool::collectAllXHits</td>
<td>5.7%</td>
</tr>
<tr>
<td>PVSeed3DTool::getSeeds</td>
<td>2.7%</td>
</tr>
<tr>
<td>PrStoreFTHit::storeHits</td>
<td>4.5%</td>
</tr>
</tbody>
</table>
Example of memory based contention

**Figure:** Table of the main CPU consumers in the application

<table>
<thead>
<tr>
<th>Function</th>
<th>Effective Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator new</td>
<td>16.7%</td>
</tr>
<tr>
<td>_int_free</td>
<td>8.3%</td>
</tr>
<tr>
<td>PrPixelTracking::bestHit</td>
<td>5.7%</td>
</tr>
<tr>
<td>PrForwardTool::collectAllXHits</td>
<td>5.7%</td>
</tr>
<tr>
<td>PVSeed3DTool::getSeeds</td>
<td>2.7%</td>
</tr>
<tr>
<td>PrStoreFTHit::storeHits</td>
<td>4.5%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Effective Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator new</td>
<td>16.7%</td>
</tr>
<tr>
<td>_int_free</td>
<td>8.3%</td>
</tr>
<tr>
<td>PrPixelTracking::bestHit</td>
<td>5.7%</td>
</tr>
<tr>
<td>PrForwardTool::collectAllXHits</td>
<td>5.7%</td>
</tr>
<tr>
<td>PVSeed3DTool::getSeeds</td>
<td>2.7%</td>
</tr>
<tr>
<td>PrStoreFTHit::storeHits</td>
<td>4.5%</td>
</tr>
</tbody>
</table>

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
- **Cycles Per Instruction**
  - 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%)
- remember Andrzej’s slide: “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 (**Structure of Arrays**) 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