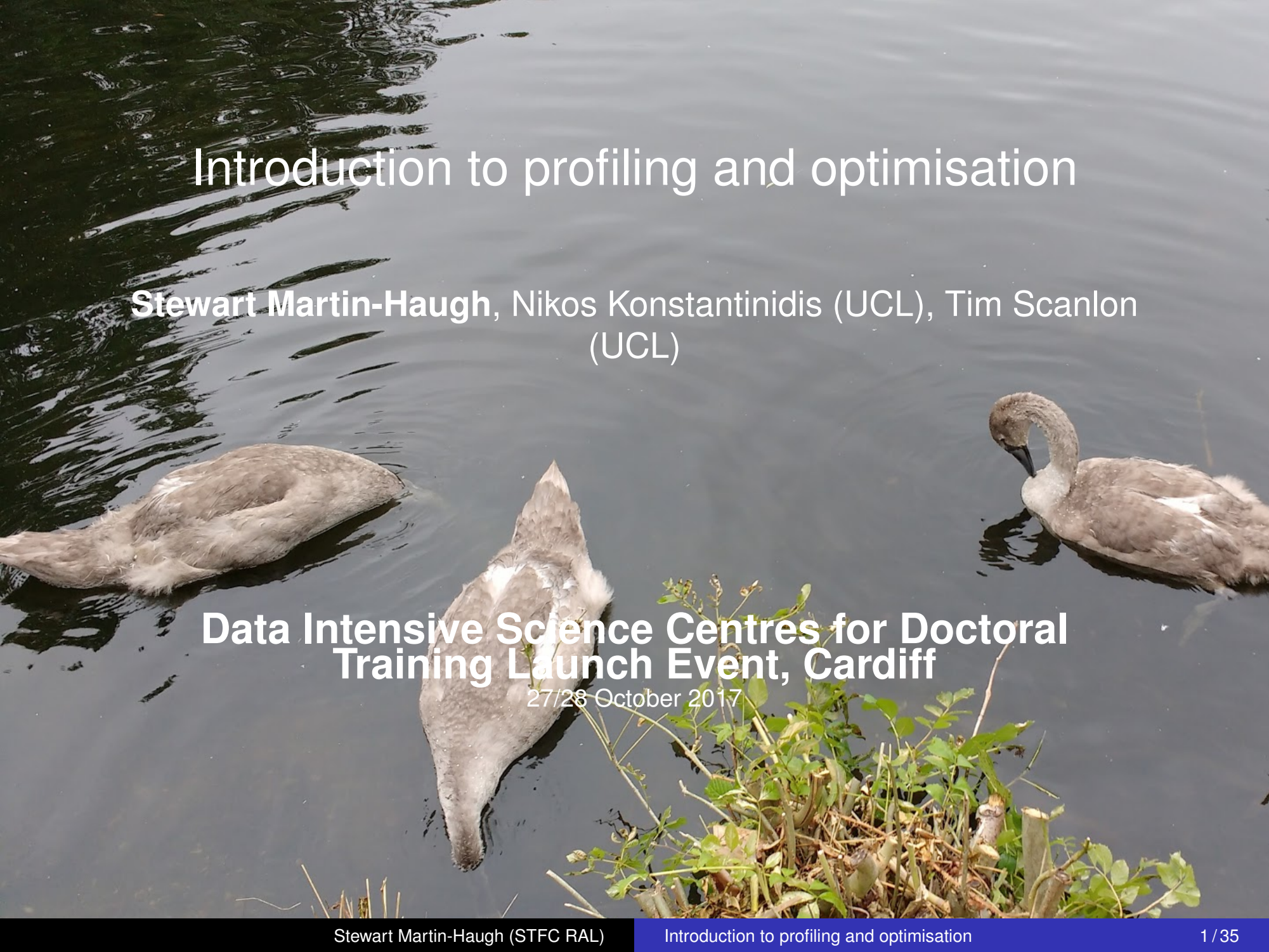


Introduction to profiling and optimisation

Stewart Martin-Haugh, Nikos Konstantinidis (UCL), Tim Scanlon (UCL)

Data Intensive Science Centres for Doctoral Training Launch Event, Cardiff

27/28 October 2017

Particle collider experiments

Large Hadron Collider is the best-known collider experiment, but there are others

- Collide protons together at high energy
- Measure fragments of proton-proton collision in specialised detectors
- Most collisions are low-energy - really interesting physics happens only 1 in a billion collisions

Data at collider detectors

Take e.g. ATLAS detector at LHC

- Each proton-proton collision record (output from all sub-detectors) $\approx$ 1 megabyte
- $\approx$ 40 million collisions every second $\rightarrow$ 40 terabyte/second output rate
- Google data storage $\approx$ 15 exabytes[1]

$$\frac{\text{15 exabytes}}{\text{40 terabytes/second}} = \text{4.34 days}$$

- Even if we fill up Google's data centres, we only produce $\approx$ 1000 Higgs boson events, and not many in "golden channels"
- In any case, we cannot read out the detector at 40 terabytes per second!

[1] https://what-if.xkcd.com/63/

Data at collider detectors

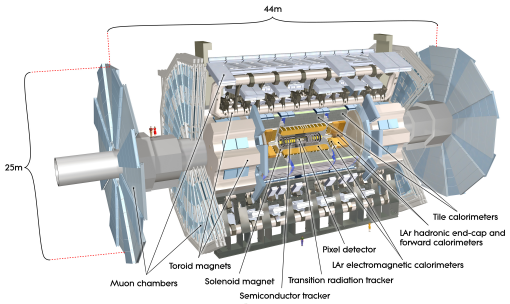
To summarise

- We physically **cannot** read out LHC data at the rate the LHC can provide
- If we could record all the data produced by the LHC, we would run out of worldwide disk space in a few days
- Once we filled up all the disk space worldwide, we still wouldn't have found the Higgs boson

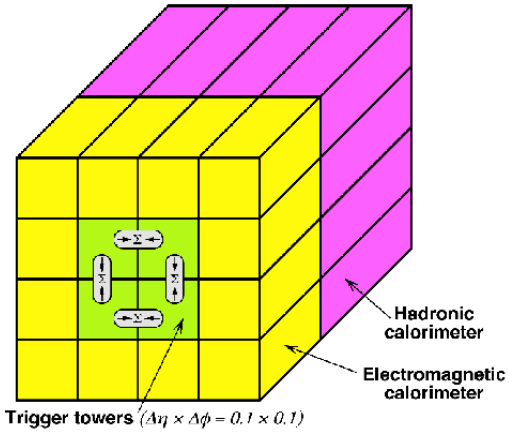
We need a **trigger** system - decide on the fly which collisions to accept and which to reject

Trigger systems for particle physics

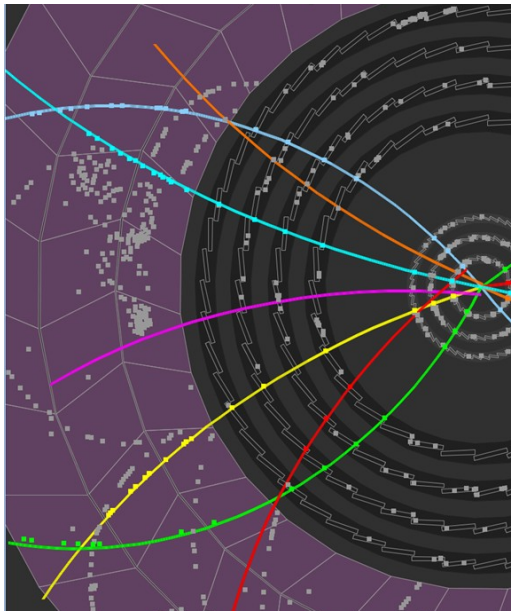
- Goal: reduce the output data rate from 40 terabytes/second to e.g. $\approx$ 1 gigabyte/second
- Need hardware and software systems that read the detector output and decide whether the collision contains interesting physics



ATLAS detector



Fast hardware - microsecond decision time



Fast software - millisecond decision time

- For a more detailed introduction, see. e.g here

Software trigger systems for particle physics

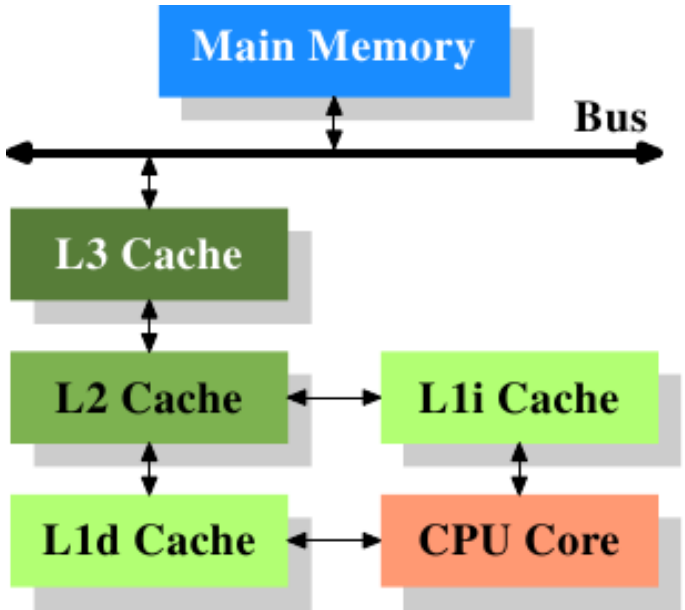
- 40,000 CPU cores needed to read collisions and reduce the data rate to an acceptable output rate (1 gigabyte/second)
- Strong requirements for trigger software
  - Fast, smart pattern recognition algorithms - excellent rejection
  - Robust software - rare/no crashes
  - Reproducible/correct software - free/mostly free of bugs, no spurious rejection of interesting physics
- Trigger systems at LHC are **the** canonical data intensive science system!
- Focus on CPU and RAM performance today
- But feel free to ask about robustness/reproducibility later

This workshop

- We assume you have an algorithm you want to use for a data intensive science application
- But you want the algorithm to run faster or use less RAM
  - Allow for more refined selection
  - Reduce computing requirements
- Need to measure and improve speed
- Focus on Linux, C++ applications

CPUs 101

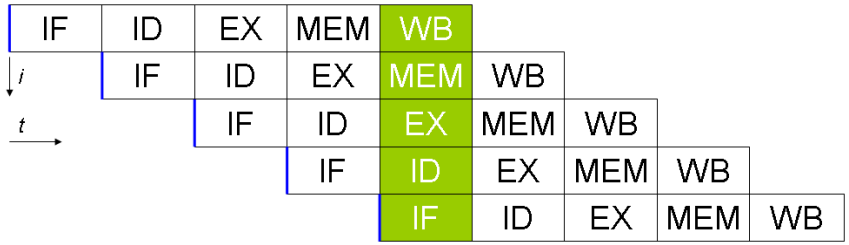
- RAM access has not improved at the same speed as CPU clocks
- Fast RAM is still very expensive - store recently used data in series of caches before checking main RAM
- Ideal case - all data found in L1 cache
- Worst case - all data retrieved from main memory



Basic cache architecture diagram from
What Every Programmer Should Know About Memory

Pipelining

- Pipelining allows processors to execute multiple instructions per clock cycle



Five stage Instruction pipeline

- Only works for linear code
- Branching (e.g. **`if`** and **`else`**) is a problem
- Naively, can only execute one instruction to test branch

Branch prediction

- Algorithm and counter within CPU to decide which branch to take (details here)
- Significant penalty if you take an unexpected branch
- See Stack Overflow discussion here

Measuring runtimes

- Basic solution: time command
- user = time spent in your code
- sys = time spent in (Linux) kernel code
- real = sum of user + sys = **Walltime**

```
>time factor
    12345678909876543211234567898765432112345678 9
12345678909876543211234567898765432112345678 9: 3 3 3 3 3
     7 300239 122516477 13357815735863 14771126268510 1
real  0m1.172s
user  0m1.162s
sys 0m0.011s
```

- You (probably) only care about user
- If you're worried about system calls, you can use strace to see which ones are used (see e.g. Julia Evans strace zine)

Measuring runtimes

- Next level in complication: debugger
- Start your program, then randomly interrupt it a few times and see which function it's in

```
^C
Thread 1 "python" received signal SIGINT, Interrupt.
0x00007f8d81f09b55 in SiSpacePointsSeedMaker_ATLxk::
    production3Sp() ()
   from libSiSpacePointsSeedTool_xk.so
(gdb) bt
#0  0x00007f8d81f09b55 in production3Sp() ()
   from libSiSpacePointsSeedTool_xk.so
#1  0x00007f8d81f0baaa in production3Sp() ()
   from libSiSpacePointsSeedTool_xk.so
#2  0x00007f8d81f0bc0b in find3Sp() ()
   from libSiSpacePointsSeedTool_xk.so
```

- This is the **callstack**
- If your program spends 90% of its time in function X, you have a 90% chance of catching it

Walltime

- Walltime is the most important number for profiling, but also the most difficult to measure accurately
- The exact problem you work on determines your reduction strategies
  - Data can be stored for later processing $\rightarrow$ “High Throughput”
  - Real-time decision $\rightarrow$ “Low Latency”

Sampling profilers

- Congratulations, you've made a basic sampling profiler!
- Sample = interrupt, look at the call stack

```
^C
Thread 1 "python" received signal SIGINT, Interrupt.
0x00007f8d81f09b55 in costlyFunction() ()
   from costlyNumerics.so
(gdb) bt
#0  0x00007f8d81f09b55 in costlyFunction() ()
   from costlyNumerics.so
#1  0x00007f8d81f0baaa in frameworkCode() ()
   from frameworkCode.so
#2  0x00007f8d81f0bc0b in main() ()
   from program.so
```

Cost

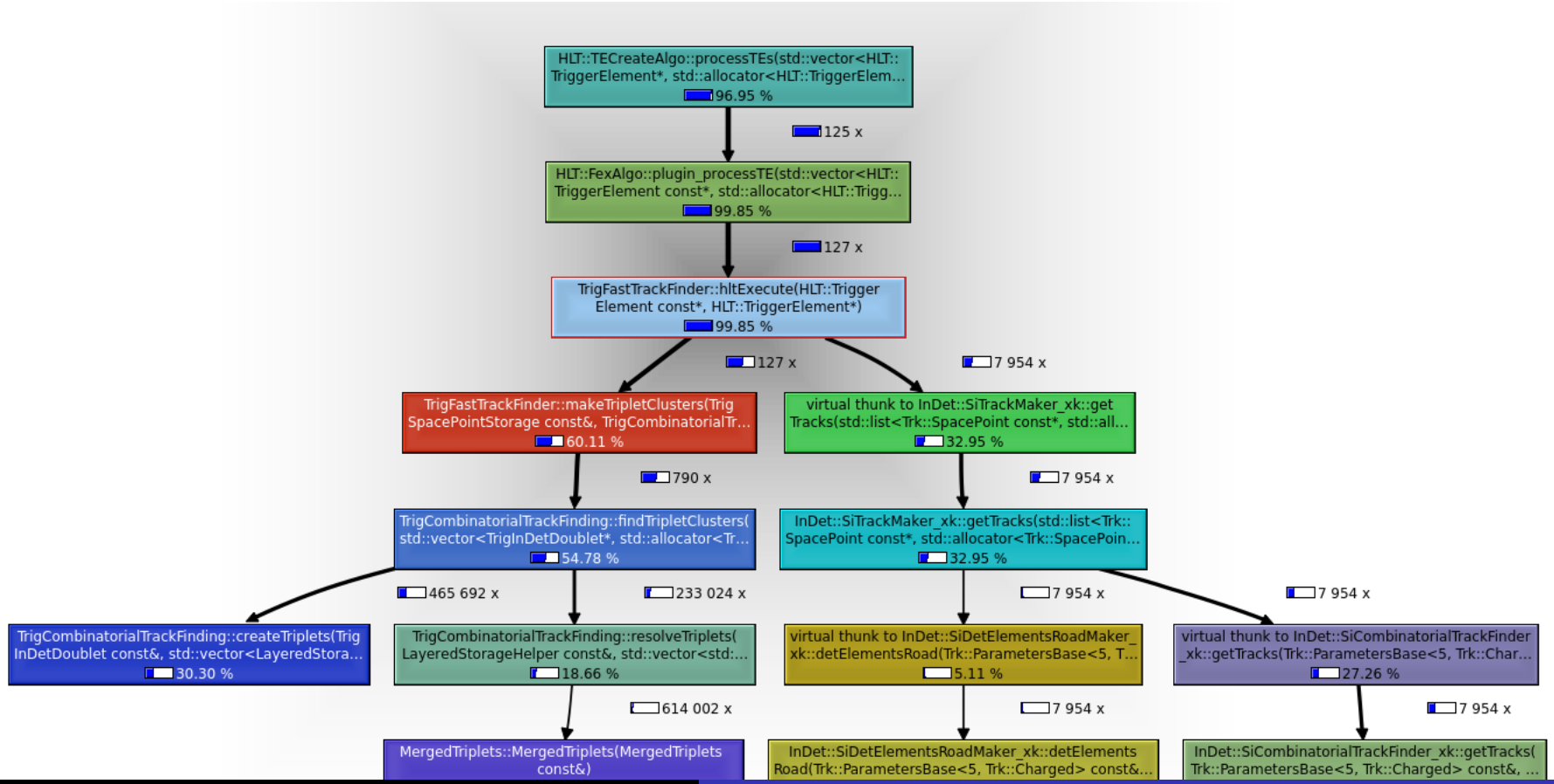
- costlyFunction() (top of the stack trace): where program was when halted
  - "Self cost"
- frameworkCall(), main(): call the function doing the work
  - "Total cost"
- Self cost $\leq$ total cost
- Focus optimisation efforts on functions with highest self-cost

```
#0  0x00007f8d81f09b55 in costlyFunction() ()
   from costlyNumerics.so
#1  0x00007f8d81f0baaa in frameworkCall() ()
   from frameworkCode.so
#2  0x00007f8d81f0bc0b in main() ()
   from program.so
```

- Some would argue this is the one true profiler

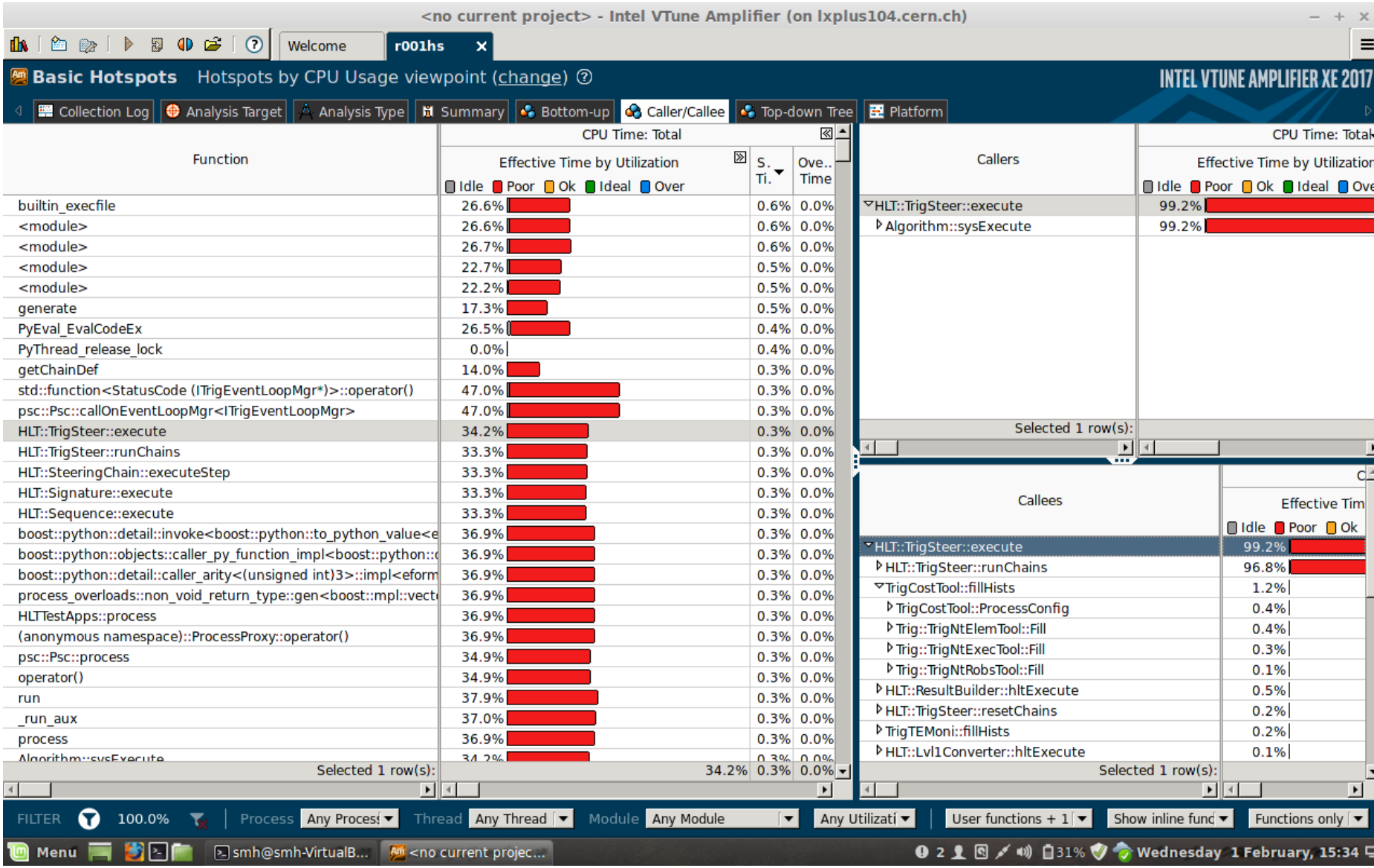
Sampling profilers

- Automate the call stack sampling procedure, generate a call graph (can be nicely explored in KCacheGrind)
- Callgrind, gperftools, Intel VTune, igprof
- Can also assign cost to lines of code (but take with a pinch of salt)



VTune

Intel VTune profiler in action



Emulation

- Valgrind has the cachegrind tool
- Emulates a basic modern CPU, with level 1, level 2 caches, branch prediction (somewhat configurable)
- Runs slowly
- Information about cache misses and branch misprediction
- Produces output suitable for KCacheGrind

Instrumentation

- **`perf`** is now the gold standard - sampling and instrumenting
- Part of Linux kernel (best results with new kernels)
- Monitor performance monitoring counters (PMCs)
- VTune also has access to these
  - Some features require root access

```
perf stat -d program
     10 152 172 182      cycles:u                    #
        3,451 GHz                         (49,86%)
     14 584 154 073      instructions:u              #
        1,44  insn per cycle              (62,43%)
      2 318 605 154      branches:u                  #
         788,130 M/sec                      (74,93%)
         44 768 463      branch-misses:u             #
            1,93% of all branches          (75,00%)
      4 116 170 377      L1-dcache-loads:u           #
         1399,150 M/sec                     (74,18%)
        167 821 302      L1-dcache-load-misses:u     #
           4,08% of all L1-dcache hits     (25,06%)
         45 252 042      LLC-loads:u                 #
            15,382 M/sec                     (24,89%)
          8 794 669      LLC-load-misses:u           #
```

Profiling thoughts

- It's a cliche, but the biggest improvements usually come from changing algorithm, not minor changes to code
- Compiler Explorer is a great tool to get a feel for what the compiler is doing to your code
- Don't second-guess modern optimising compilers
- Measure and benchmark
  - Heard good things about Google Benchmark, but not used it yet

CPU optimisation

- Once you've identified which part of your code takes the most time, you can start optimising
- Strategies are somewhat language-dependent, but some general points always true
- Compiled languages (C++, Fortran) faster than interpreted (Python, Ruby)
- Standard libraries (FFTW, BLAS, Eigen) likely faster than your own code - don't reinvent the wheel!

Loops

- Don’t recalculate the same values within loops: move code outside

Higher mathematical functions

- Trigonometric functions are slow
  - Consider using an optimised library (see e.g. here)
- For linear algebra, definitely use a library (e.g. Eigen

Floating-point operations

- Avoid square root if possible - classic examples

```
float r = x*x + y*y;
if (sqrt(r) < max_r) {
  return true;
}

if (r < max_r_squ) {
  return true;
}
```

- Avoid division if possible

```
y=x/5.0;
y=x*0.2;
```

- Compiler can't do this for you (exception: -ffast-math option)

Branching

- Don’t recalculate within loops: move code outside

Putting it all together

- Sometimes you can remove branches and reduce number of operations all at once

```
//OLD
if ( h >= 0.) {
  h = min( max( 0.25*h, pow((x / y), 0.25)*h), 4.*h);
} else {
  h = max( min( 0.25*h, pow((x / y), 0.25)*h), 4.*h);
}
//NEW
h = h*min( max( 0.25, pow((x / y), 0.25)), 4.);
```

Algorithmic complexity

- If possible, stick to standard algorithms (e.g. C++ std::sort) instead of writing your own
- If the algorithm is a hotspot, consider trying out different algorithms (note, flashing lights)

Data structures

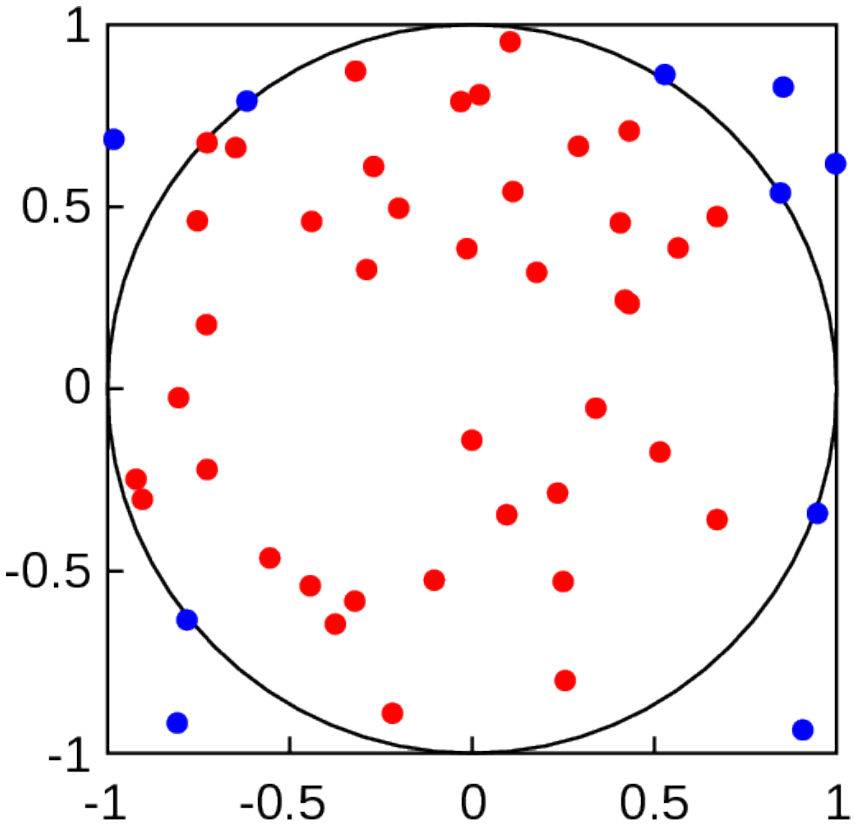
- Worth thinking about which data format fits your problem
- In C++, **`std::vector`** is probably a good fit

Points to remember

- Profiling and reasoning about code cannot tell you if you're using the wrong algorithm
- Writing your own implementation of something is an excellent way to learn, even if you never use it

Exercise 1

- Classic demonstration of Monte Carlo integration
- Throw darts at a circle inscribed in a square
- Ratio of darts hitting the blackboard tends to $(\pi r^2)/(4r^2) = \pi/4$
- Exercise_1/exercise_1.cpp code



Memory profiling

- Using too much memory is bad for two reasons
  - Eventually you run out (e.g. memory leak)
  - Allocating memory has a significant CPU cost - higher if your data doesn't fit in e.g. L1 cache
  - A single large allocation is cheaper than several small allocations

Different allocators

- Your program will not just receive the memory it asks for when it asks for it
- Allocator decides how much to request at a time and how much should be contiguous
- **`glibc`** used by default
- Others available, particularly **`jemalloc`** (Facebook) and **`tcmalloc`** (Google)
- No need to recompile, just preload
- May work better for your memory access pattern than glibc

Finding big allocations

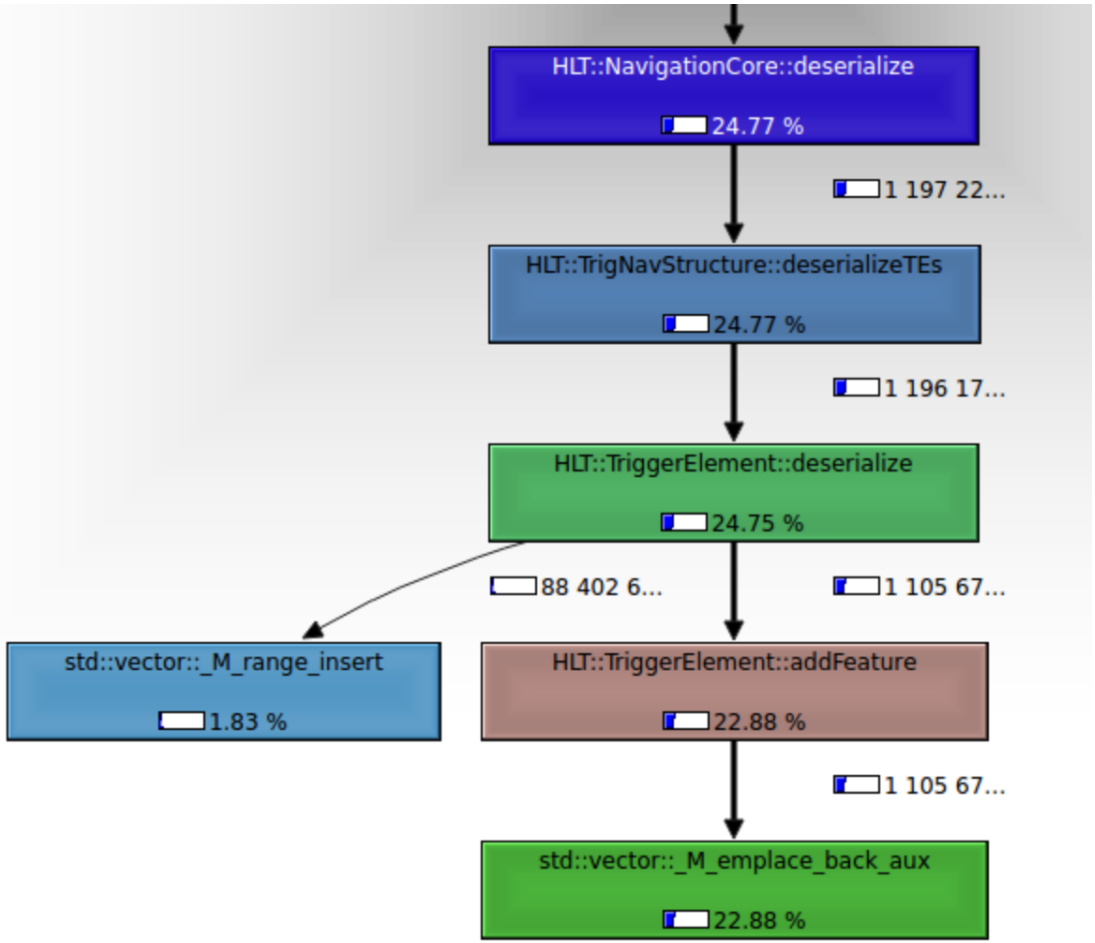
- Scenario: your program is running out of memory
- How to track down large (e.g. 1 GB) allocations?
- **`tcmalloc`** provides a printout when this happens

```
tcmalloc: large alloc 2720276480 bytes == 0x73eda000 @
tcmalloc: large alloc 2720276480 bytes == 0x2a96f0000 @
tcmalloc: large alloc 2720276480 bytes == 0x34b932000 @
```

- Add a breakpoint at ReportLargeAlloc

Heap profilers

- **`jemalloc`** and **`tcmalloc`** both come with low-overhead profilers to analyse which functions allocate most memory
- Output can be interpreted much as with a call-graph



Leak checkers

- Valgrind is the workhorse here
- Will not find a leak allocated towards the end of runtime
- Slow, high memory requirements (but requires no recompilation)