14th Inverted CERN School of Computing (2023)

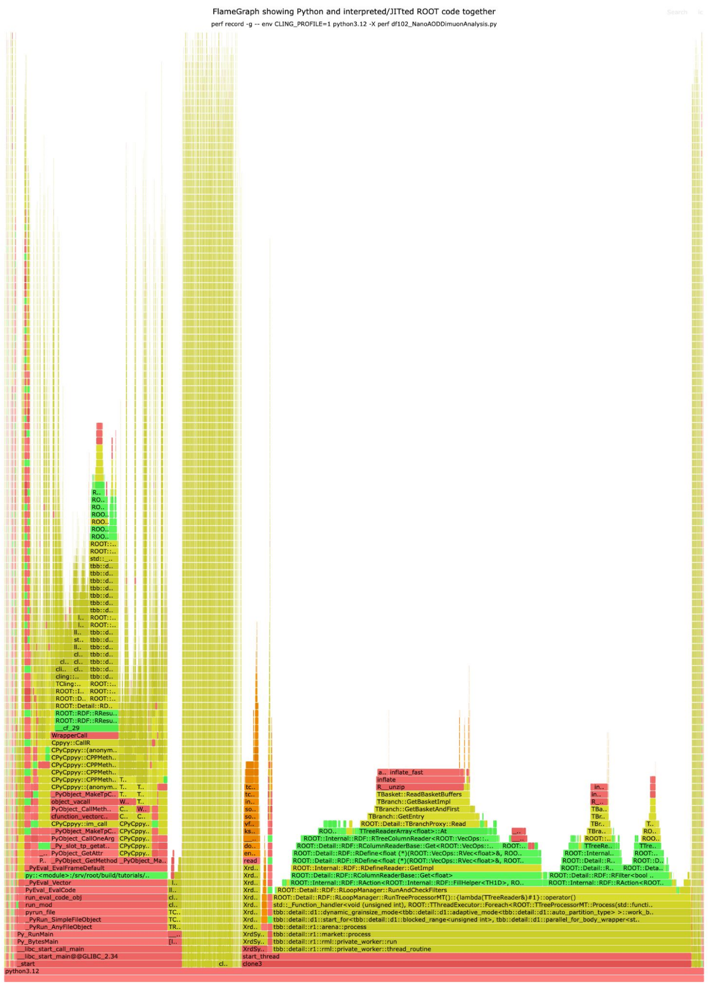
CPU Performance Profiling on Linux in the HEP Context

Ivan Donchev Kabadzhov

09/03/2023

Goals of this lecture

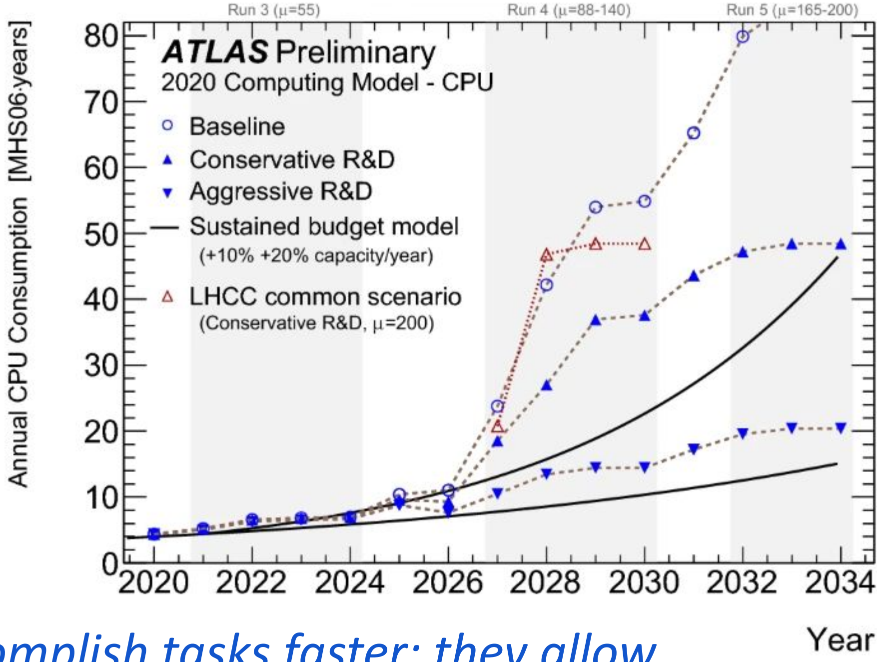
- Familiarize with several common performance bottlenecks - e.g. core bound, memory bound, etc.
- Use performance analysis tools to identify hotspots:
  - `perf stat`
  - `perf record`
  - `perf report`
- Interpret CPU flame graphs



Introduction

Motivation

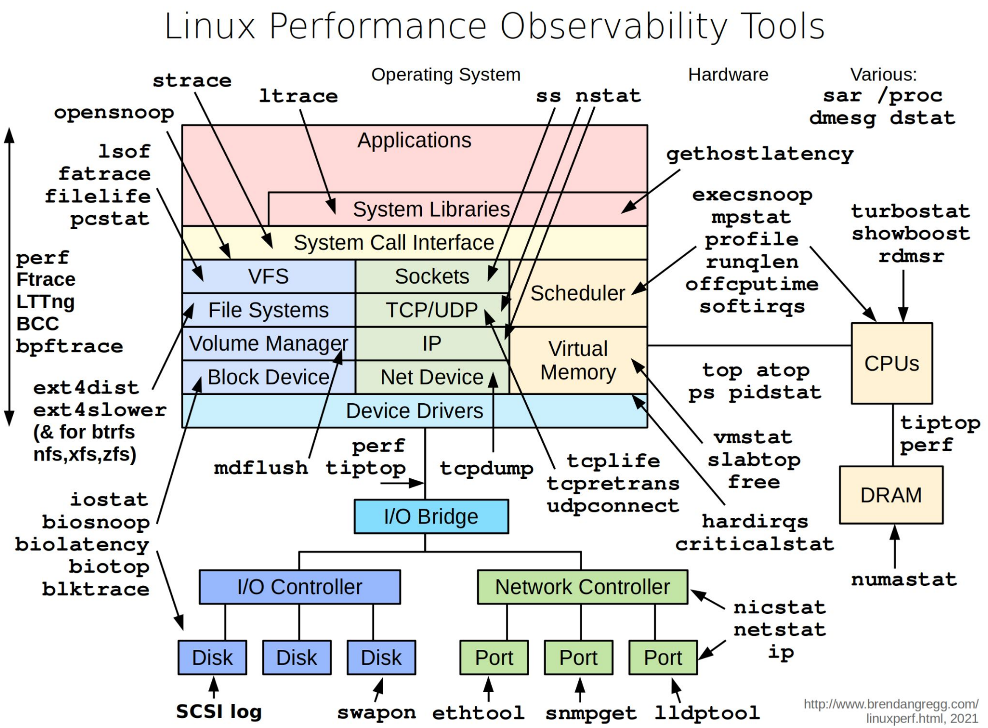
- Problem: If experiments preserve their computing models, they will be very soon CPU bound
- Finding performance bottlenecks is usually difficult:
  - Complicated CPU architectures
  - Overhead from measurements
  - Missing symbols and stack frames
  - Concurrency issues



“Fast tools don’t just allow users to accomplish tasks faster; they allow users to accomplish entirely new types of tasks, in entirely new ways” [src]

Performance Analysis Processes

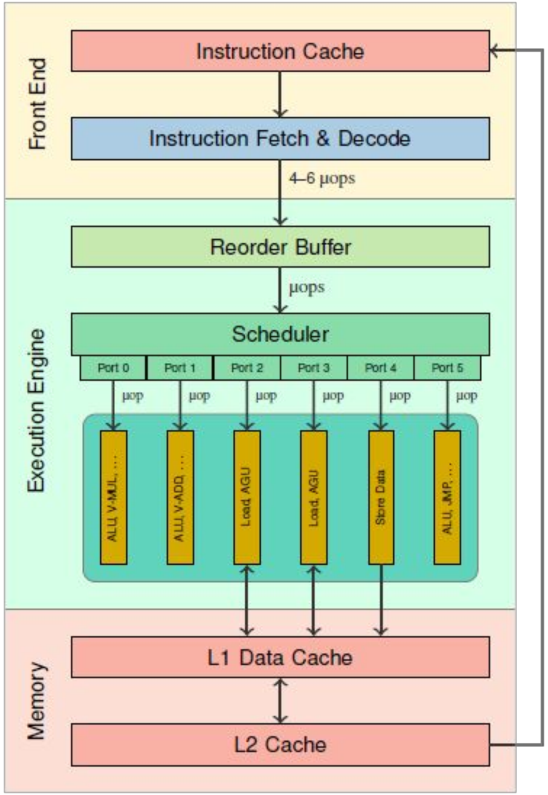
- System Level:
  - Kernel
  - Disk
  - Network
- Application Level:
  - CPU Utilization
  - Algorithmic complexity
- Hardware Level:
  - Cache misses
  - Branch mispredictions
  - Data dependencies



http://www.brendangregg.com/linuxperf.html, 2021

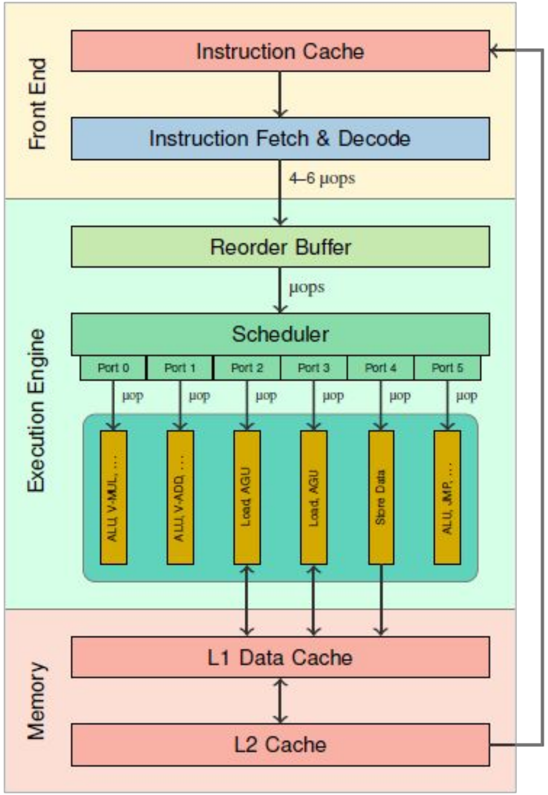
Pipeline of the CPU

- Front end:
  - fetches instructions from the memory
  - decodes them into microoperations (µops)
- Back end (execution engine):
  - Reorder buffer:
    - stores the the µops until they retire
    - allocates and renames registers
  - Scheduler:
    - queues the µops until all their source operands are ready ⟶ dispatches them through an execution port



Common Problems in the Pipeline

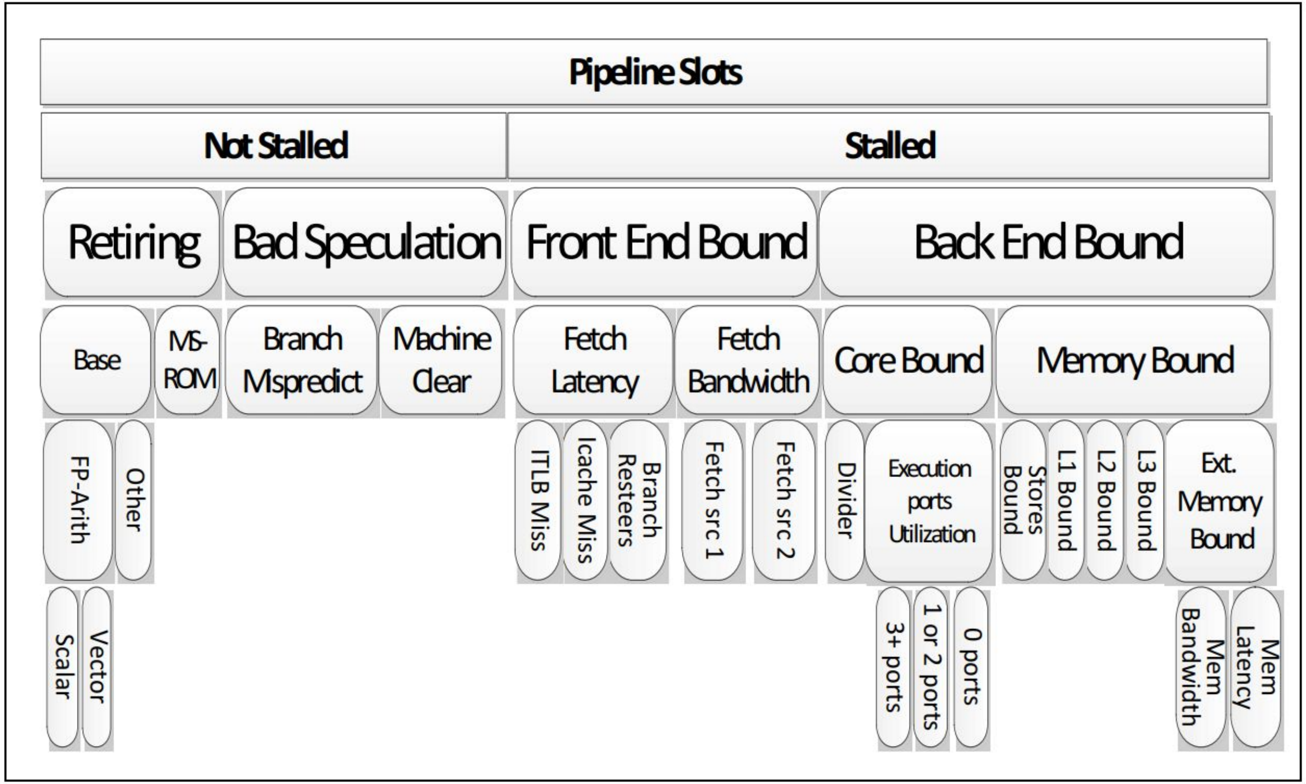
- Front end:
  - Code Duplication
  - Code Layout (Locality)
  - Frequent Branching
- Back end (execution engine):
  - Core Bound:
    - Data Dependencies
    - Divisions and Special Functions
  - Memory Bound
    - False Sharing
    - Scattered Memory Accessing



Top-down Microarchitecture Analysis

- During any cycle, a pipeline slot can either be empty or filled with a uOp.
- If a **slot is empty** during one clock cycle ⇒ attribute this to a **stall** ⇒ check if the stall is due to Front-end or Back-end
  - Front-end's inability to fill the slot with a uOp ⇔ **Front-End Bound**
  - Back-end is not ready to handle Front-end's uOp ⇔ **Back-End Bound**
- If the **processor is not stalled** then a pipeline slot will be filled with a uOp at the allocation point
  - If the uOp eventually retires ⇔ **Retiring**
  - If it does not retire ⇔ an incorrect branch prediction by the Front-end or a clearing event, i.e. pipeline flush ⇔ **Bad Speculation**

Top-down Microarchitecture Analysis



CPU Performance Profiling

Stack Traces

Code path snapshot, e.g. from **gdb** (C++) and **pdb** (Python)

C++

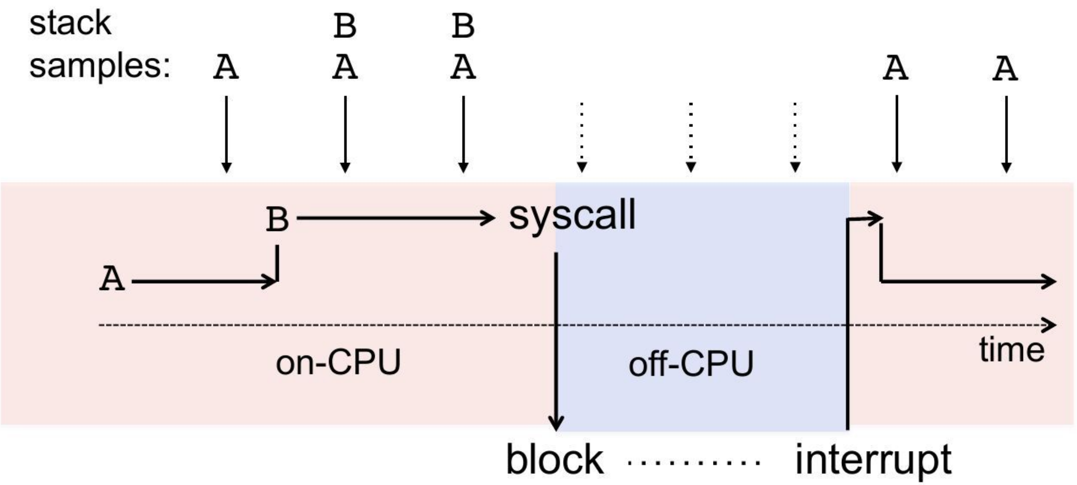
```
$ gdb ./exec
>>> bt
#0 c () at test.cpp:4
#1 0x30d9 in b () at f.cpp:8
#2 0x30e9 in a () at g.cpp:12
#3 0x3104 in main () at M.cpp:16
```

```
$ python -m pdb test.py
(Pdb) where
  t.py(12)<module>()
-> a()
  t.py(9)a()
-> return b() ...
```

Idea of CPU Profiling

Record stacks at a timed interval

- Pros: Low (deterministic) overhead
- Cons: Coarse accuracy, but usually efficient



Perf

- Is the de-facto-standard profiling infrastructure on Linux
- Provides access to performance data including operating system software events and hardware performance events
- Supports counting, sampling (and also tracing) mode
- Includes tools to collect, display and analyze performance data
- Answers the questions:
  - Which parts of the program take the most execution time?
  - Do software or hardware events indicate an actual performance issue?
  - Which code-paths are causing CPU level 2 cache misses?
  - Are the CPUs stalled on memory I/O?
- But does not answer the question: How to fix the performance issue?

Perf

Cherry-picked some interesting (for this presentation) perf tools:

```
$ perf
usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]
The most commonly used perf commands are:
     diff       Read perf.data files and display the differential profile
     list       List all symbolic event types
     record     Run a command and record its profile into perf.data
     report     Read perf.data (created by perf record) and display the profile
     script     Read perf.data (created by perf record) and display trace output
     stat       Run a command and gather performance counter statistics
See 'perf help COMMAND' for more information on a specific command.
```

perf_events

- perf_events instruments "events", which are a unified interface for different kernel instrumentation frameworks. Events can be collected, include timestamps, code paths, etc. The types of events include:
  - Hardware Events: CPU performance monitoring counters. For instance:
    - `cpu-cycles`
    - `cache-references, cache-misses`
    - `branch-misses`
  - Software Events: These are low level events based on kernel counters
    - `page-faults`
    - `context-switches`
- See all perf_events with `perf list` and enable them with `-e`

Cookbook for Profiling

1. `perf stat` - Check the rate of events to roughly estimate the volume of data you will be capturing
2. `perf record` - Run the program and collect profile data
3. `perf report` - Analyze the data and display the profile

Sidebar: Program under investigation

- We will go through a simple (C++ compiled) HEP analysis in RDF
- Basic idea: We have a columnar data in a ROOT file. We will be filtering data based on some interesting events and create new columns.
- Functionality: Plot the pt of the tri-jet system with mass closest to 172.5 GeV, and the leading b-tag discriminator among the 3 jets in the triplet [requires looping on combination of objects in the same collection, and extracting properties of a combination other than the key used to sort them] (src, benchmark 6)
- The exact example is at https://github.com/ikabadzhov/iCSC-Profiling

perf stat

```
$ perf stat -r3 ./rdf args  # -r3 => 3 runs, get deviations as well (superior to time)
 Performance counter stats for './rdf args' (3 runs):
          37480.10 msec task-clock                #    1.002 CPUs utilized          ( +-  0.14% )
               139      context-switches          #    3.717 /sec                   ( +- 10.86% )
                 7      cpu-migrations            #    0.187 /sec                   ( +- 17.17% )
            335844      page-faults               #    8.982 K/sec                  ( +-  7.03% )
      167819805846      cycles                    #    4.488 GHz                    ( +-  0.15% )
      346334459275      instructions              #    2.07  insn per cycle         ( +-  0.04% )
       55620973671      branches                  #    1.488 G/sec                  ( +-  0.05% )
        1155592432      branch-misses             #    2.08% of all branches        ( +-  0.13% )
           37.3921 +- 0.0525 seconds time elapsed  ( +-  0.14% )
```

- Often used to get initial clue on the application under investigation
- Can also bind this tool to a specific process or thread (`-p`/`-t`)
- Can collect samples from specific set of CPUs only (`-C`)
- Can measure a specific set of interesting events/metrics (`-e`/`-M`)

perf stat -M TopdownL1 (Intel only)

```
$ perf stat -M TopdownL1 ./rdf args # alternatively `--topdown -a`
 Performance counter stats for './rdf original input/ 30':

       6938004364      INT_MISC.RECOVERY_CYCLES         #     0.16 Bad_Speculation          (49.99%)
     168185520160      CPU_CLK_UNHALTED.THREAD          #     0.50 Retiring                 (49.99%)
     338892912067      UOPS_RETIRED.RETIRE_SLOTS                                            (49.99%)
     415591157317      UOPS_ISSUED.ANY                                                      (49.99%)
      78717271802      IDQ_UOPS_NOT_DELIVERED.CORE      #     0.12 Frontend_Bound
                                                 #     0.22 Backend_Bound            (50.02%)
       7070038631      INT_MISC.RECOVERY_CYCLES                                             (50.02%)
     168136953070      CPU_CLK_UNHALTED.THREAD                                              (50.02%)
     415194423171      UOPS_ISSUED.ANY                                                      (50.02%)

     37.586720560 seconds time elapsed

     36.663338000 seconds user
      0.894419000 seconds sys

     53,331883054 seconds time elapsed
```

perf record

```
$ perf record -g -o df.data --call-graph=fp -F99 ./df args
[ perf record: Woken up 165 times to write data ]
[ perf record: Captured and wrote 42.086 MB original.data (5201 samples) ]
```

- `--call-graph=fp` - uses frame pointers (and is the default)
- `--call-graph=dwarf` - records (user) stack dump (a small study on it)
- `-F99` - profiling frequency (in Hertz)
- Can also bind this tool to a specific process or thread (`-p`/`-t`)
- Can collect samples from specific set of CPUs only (`-C`)
- Can collect samples from a specific event, default is cycles (`-e`)

perf report

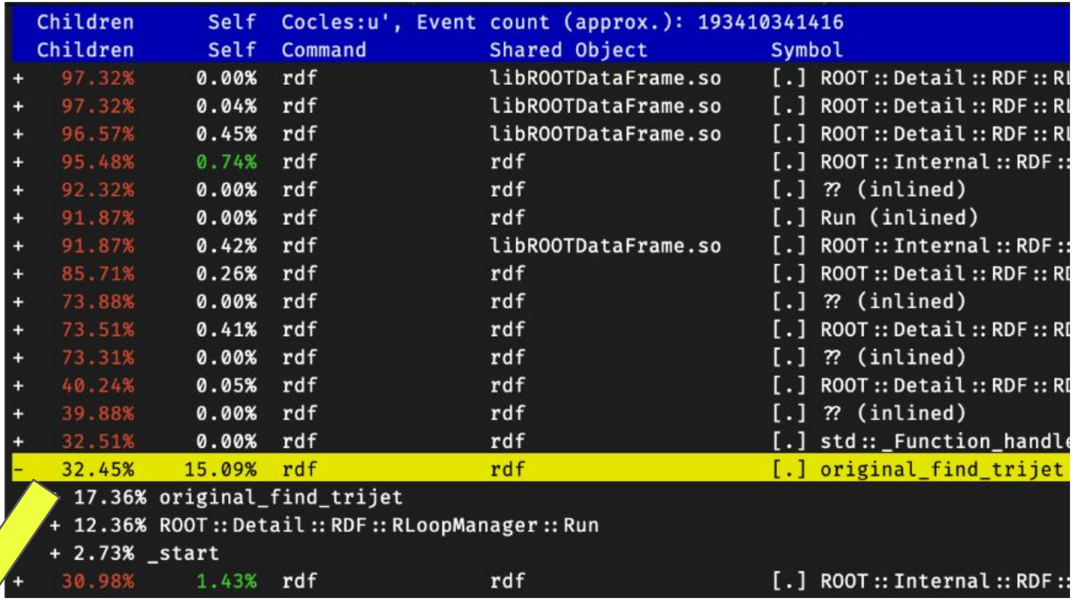
Caveat: Read only .data files from the same machine

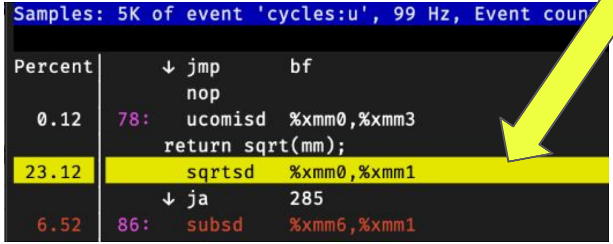
```
$ $ perf report -i original.data --stdio | awk '(NR >= 4 && NR <= 9) || (NR >= 12 && NR <= 20)'
# Total Lost Samples: 0
#
# Samples: 5K of event 'cycles:u'
# Event count (approx.): 193410341416
#
# Children      Self  Command          Shared Object          Symbol
    97.32%     0.00%  rdf              libROOTDataFrame.so    [.]
ROOT::Detail::RDF::RLoopManager::Run
            |
            ---ROOT::Detail::RDF::RLoopManager::Run
               ROOT::Detail::RDF::RLoopManager::RunTreeReader
               |
               |--96.55%--ROOT::Detail::RDF::RLoopManager::RunAndCheckFilters
               |          |
               |          |--91.87%--Run (inlined)
               |          |          ?? (inlined)
```

Usually, more useful not to pass `--stdio`, see next slide

perf report

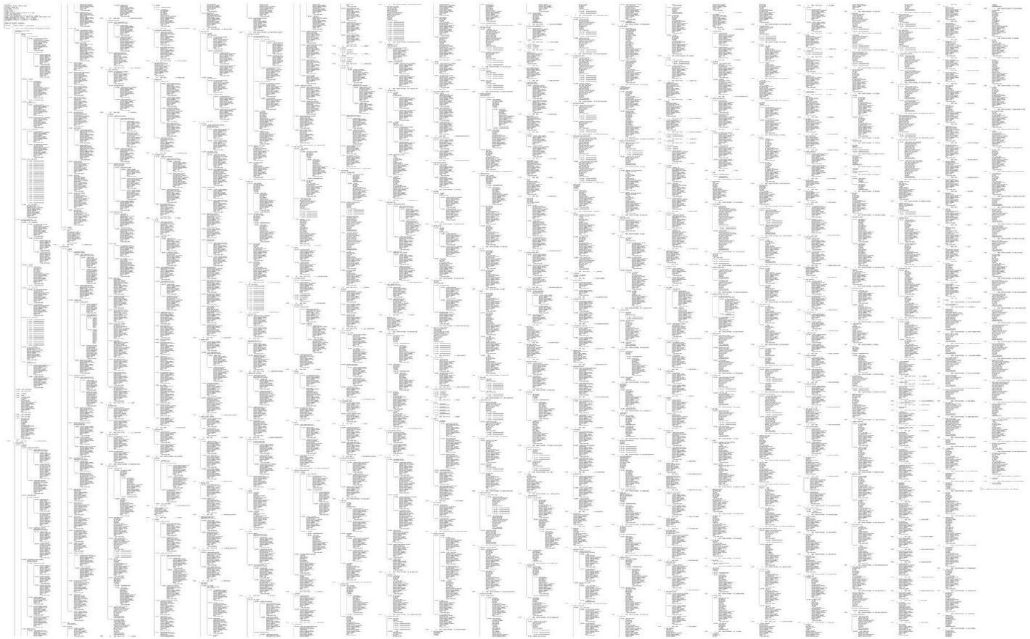
- Expand/Collapse (`+`/`-`/`e`)
- Filter symbol by name (`/`)
- Can show actual functions and instructions in the object code (annotate). Press `a` in the report
- And a lot more (see with `h`)





- `H` → go the hottest instruction
- `tab` → go to the next hottest

Problems of the perf report



- Sometimes hard to make sense from the output
- Broken stack frames

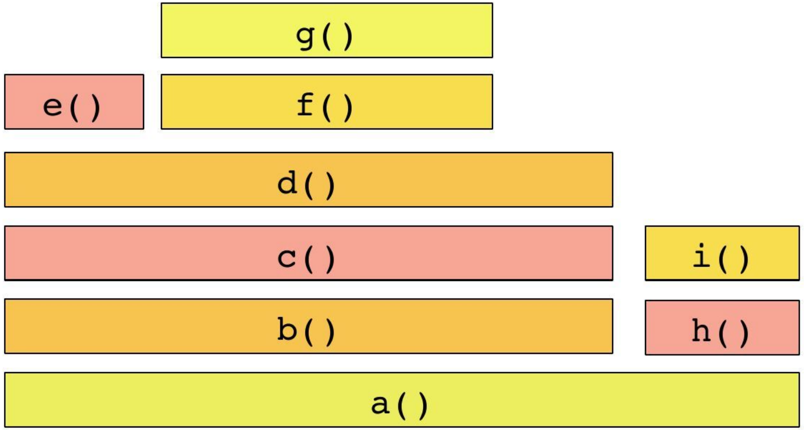
```
[unknown]
?? (inlined)
??
```

Solution 1: Fixing broken frames

- If working on a remote machine ⇒ ask admins for more access privileges!
- C++:
  - Compile the software with: **`-O2 -g -fno-omit-frame-pointer`**
  - Inlined frames are okay, they were likely optimized from `-O2`
  - When relying on (external) frameworks → **build them from source**
  - Enable debugging information for glibc (src, but use with care!)
    `LD_LIBRARY_PATH=/usr/lib/debug`
- Python (>=3.12, since 2022):
  - Need to tell Python that profiling support is enabled, 2 ways:
  - Set an environment variable PYTHONPERFSUPPORT=1
  - Use the **`-X perf`** option, i.e. `python -X perf script.py`

Solution 2: Flame Graphs

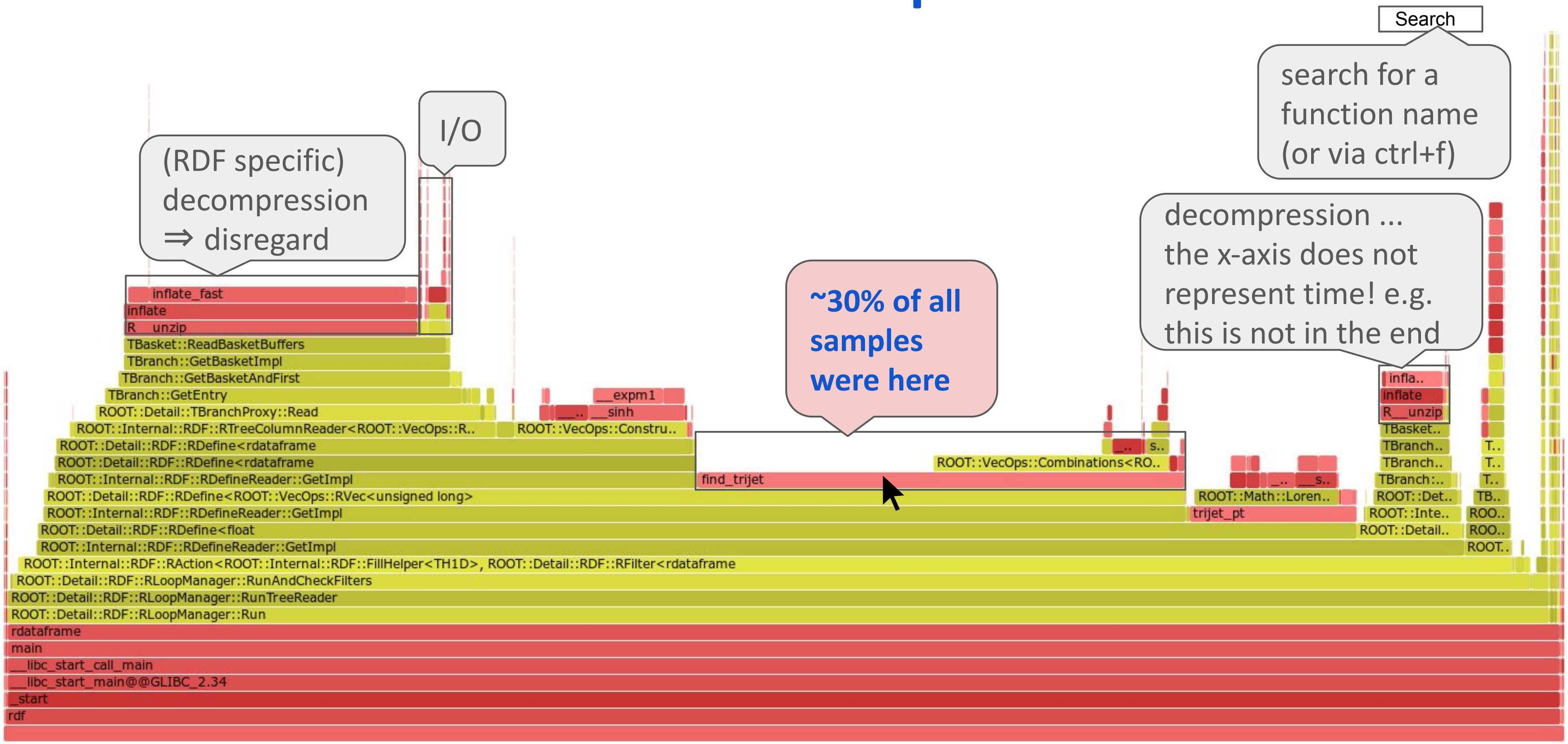
- Visualizes a collection of stack traces:
  - Top edges - who is running on-CPU
  - Width - number of samples
  - Top-down - call stack
  - Custom color codes, i.e. Java colors:
    - Red - user level
    - Orange - kernel
    - Yellow - C++



- Get stackcollapse-perf.pl and flamegraph.pl from here and then:

```
$ perf script -i original.data | ./stackcollapse-perf.pl | \
  ./flamegraph.pl -w 1500 --colors java > original.svg
```

Solution 2: Flame Graphs



Function: original_find_trijet (52,598,984,053 samples, 31.19%)

Optimization

Understanding “What is Going On”

```
... original_find_trijet(Vec<XYZTVector> &jets) {
 const auto c = ROOT::VecOps::Combinations(jets, 3);
 float distance = 1e9; const auto top_mass = 172.5;
 for (auto i = 0u; i < c[0].size(); i++) {
   auto p1 = jets[c[0][i]];
   auto p2 = jets[c[1][i]];
   auto p3 = jets[c[2][i]];
   const auto tmp_mass = (p1 + p2 + p3).mass();
   const auto tmp_distance = std::abs(tmp_mass -
top_mass);
   if (tmp_distance < distance) {
     distance = tmp_distance; idx = i;
 }}
 return {c[0][idx], c[1][idx], c[2][idx]};}
```

- The Combinations function returns the indices that represent all unique combinations of elements, i.e.: `v=RVecD{-0.5,3.14,42.};` `Combinations(v,2);` ⇒ `{{0,0,1}, {1,2,2}}`
- Do we need to create a 2-dimensional vector to represent all combinations?

Understanding “What is Going On”

So many `-e instructions`!

```
... original_find_trijet(Vec<XYZTVector> &jets) {
 const auto c = ROOT::VecOps::Combinations(jets, 3);
 float distance = 1e9; const auto top_mass = 172.5;
 for (auto i = 0u; i < c[0].size(); i++) {
   auto p1 = jets[c[0][i]];
   auto p2 = jets[c[1][i]];
   auto p3 = jets[c[2][i]];
   const auto tmp_mass = (p1 + p2 + p3).mass();
   const auto tmp_distance = std::abs(tmp_mass -
top_mass);
   if (tmp_distance < distance) {
     distance = tmp_distance; idx = i;
 }}
 return {c[0][idx], c[1][idx], c[2][idx]};}
```

```
Samples: 5K of event 'instructions:u', 99 H
Percent │    ↓ jae       188
        │    auto p1 = jets[c[0][i]];
  8.28  │bf:   mov       (%rdi),%rax
        │    return begin()[idx];
        │      mov       0x0(%r13),%rdx
  0.10  │      mov       (%rax,%r12,8),%rcx
        │    auto p2 = jets[c[1][i]];
        │      mov       0x50(%rdi),%rax
  5.76  │      mov       (%rax,%r12,8),%rsi
        │    auto p3 = jets[c[2][i]];
  0.10  │      mov       0xa0(%rdi),%rax
        │      shl       $0x5,%rcx
  0.31  │      add       %rdx,%rcx
  6.66  │      mov       (%rax,%r12,8),%rax
        │      shl       $0x5,%rsi
        │      add       %rdx,%rsi
        │      shl       $0x5,%rax
```

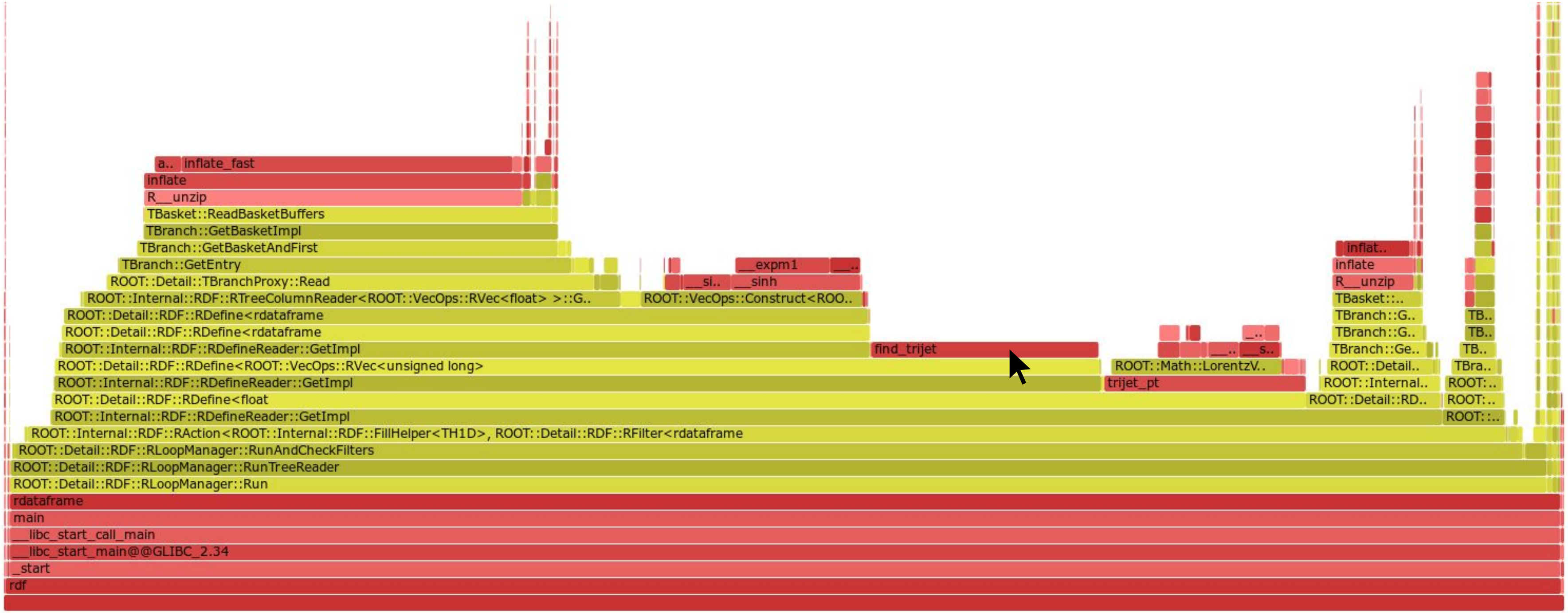
Improved Version

```
... direct_find_trijet(Vec<XYZTVector> &jets) {
 float distance = 1e9; const auto top_mass = 172.5;
 std::size_t idx1 = 0, idx2 = 1, idx3 = 2;
 const auto n_jets = jets.size();
 for (std::size_t i = 0; i < n_jets - 2; i++) {
   auto p1 = jets[i];
   for (std::size_t j = i + 1; j < n_jets - 1; j++) {
     auto p2 = jets[j];
     for (std::size_t k = j + 1; k < n_jets; k++) {
       auto p3 = jets[k];
       const auto tmp_mass = (p1 + p2 + p3).mass();
       const auto tmp_distance = std::abs(tmp_mass - top_mass);
       if (tmp_distance < distance) {
         distance = tmp_distance; idx1 = i; idx2 = j; idx3 = k;
 }}}} return {idx1, idx2, idx3}; }
```

- No need to allocate extra memory to store combinations
- Sequential memory access only

Warning! Ensure that the optimized code is **correct**!

Flame Graph of the Improved Version



Function: direct_find_trijet (19,244,401,542 samples, 14.55%)

Takeaways:

- You should **not guess** why the execution of a program is slow
- Instead make use of **profiling tools to understand** why the program is slow
- In particular, you should now know how to analyze performance with:
  - `perf stat`
  - `perf record/report`
  - Flame graphs
- **Man pages** are your best friends: perf, perf stat, perf record, perf report
- Last but not least, another very power profiling tool is VTune (Intel only)

References and Very Useful Links

- Perf examples - https://www.brendangregg.com/perf.html
- Some presentations and articles on the perf profiling topic:
  - https://indico.cern.ch/event/1177921/
  - https://files.gotocon.com/uploads/slides/conference_60/2394/original/YOW2022_flame_graphs.pdf
  - http://sandsoftwaresound.net/perf/perf-tutorial-hot-spots/
  - https://easyperf.net/categories/#performance%20analysis
- Intel's Top-down Microarchitecture Analysis Cookbook - https://www.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/methodologies/top-down-microarchitecture-analysis-method.html
- To go through the example - https://github.com/ikabadzhov/iCSC-Profiling

And sincere thanks to
Guilherme Amadio
&
Enrico Guiraud
for their guidance
&
Giovanna Lazzari Miotto
for her help producing the benchmarks

Questions?

Backup Slides

Use Case: Version Differences

- General workflow:
  - Code ⟶ Debug ⟶ Code ⟶ Measure/Profile
  - Optimize ⟶ Debug ⟶ Optimize ⟶ Measure/Profile
- Want to check how the new performance profile compare to the old one
  - Poor man's solution: read reports and flame graphs from different tabs
  - `perf diff` - Display a differential profile (similar to perf report)
  - Differential flamegraphs (src):
    - Red - more samples (color intensity - degree of difference)
    - Blue - less samples (color intensity - degree of difference)

```
$ perf script -i original.data | ./stackcollapse-perf.pl > out.folded1
$ perf script -i direct.data | ./stackcollapse-perf.pl > out.folded2
$ ./difffolded.pl out.folded1 out.folded2 | ./flamegraph.pl -w 1500 > d.svg
```

Another Direction of Optimization

- A “classical example” in profiling is matrix multiplication
- It is very important the understand the memory access of the program and choose the optimal one

- In the original case, using Combinations, recall that:
  `v=RVecD{-0.5,3.14,42.};`
  `Combinations(v,2);`
  ⇒ `{{0,0,1}, {1,2,2}}`
- What would be the implications, if the returned result is instead:
  `{{0,1}, {0,2}, {1,2}}?`
- You can try it out on github - see `transposed_find_trijet`!

Digging the Example Further

- It easy to see that the initial version of the find_trijet has complexity:
  - O(N^3) time (it is precisely N choose K=3, convince yourself why)
  - O(N^3) space (same reasoning as above)
- It is also easy to see that the latter version has complexity:
  - O(N^3) time (still brute-force go through all combinations)
  - O(1) space (no extra space allocations!)
- In particular, the underlying problem in find_trijet is to find a combination of 3 distinct elements of the input vector, so that the mass of the sum has a value closest to a const value ⇒ version of the 3Sum Closest problem (O(N^2) time)
- However, I could not make the solution of the latter problem work here, since p1.mass() + p2.mass() is different than (p1 + p2).mass()! Think why it matters!

Digging the Example Further

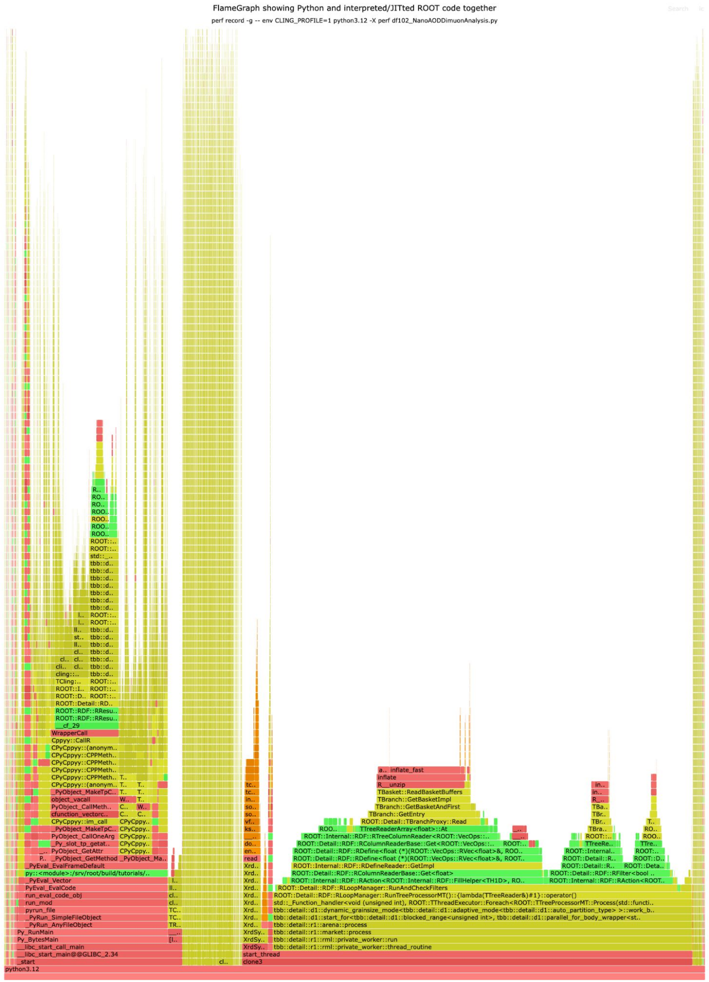
```
... nsquare_find_trijet(Vec<XYZTVector> &jets) {
 float distance = 1e9; const auto top_mass = 172.5;
 std::size_t idx1 = 0, idx2 = 1, idx3 = 2;
 ROOT::RVec<std::size_t> inds(jets.size()); std::iota(inds.begin(), inds.end(), 0);
 std::sort(inds.begin(), inds.end(), [&jets](const auto &a, const auto &b) {
   return jets[a].mass() < jets[b].mass(); });
 for (std::size_t i = 0; i <= jets.size() - 2; ++i) {
   std::size_t j = i + 1, k = jets.size() - 1;
   while (j < k) {
     const auto tmp_mass = (jets[inds[i]] + jets[inds[j]] + jets[inds[k]]).mass();
     if (tmp_mass == top_mass)
       return {inds[i], inds[j], inds[k]};
     const auto tmp_distance = std::abs(tmp_mass - top_mass);
     if (tmp_distance < distance) {
       distance = tmp_distance; idx1 = inds[i]; idx2 = inds[j]; idx3 = inds[k]; }
     if (tmp_mass < top_mass) ++j;
     else --k;
 }}
 return {idx1, idx2, idx3}; }
```

ROOT/RDF Jitted Code

ROOT/RDF (see this): Extra complication - **JIT**-ting!

On top of the regular set of actions to profile, need all steps below:

- Build ROOT from source with extra flags to read jitted symbols
- Add extra Perf permissions (`kernel.perf_event_paranoid=-1`)
- Set environment variable `CLING_PROFILE=1`
- Demangle the jitted symbols in the produced .data file



Flame graph by: Guilherme Amadio