AdaptivePerf: a profiler for single- and multi-threaded applications

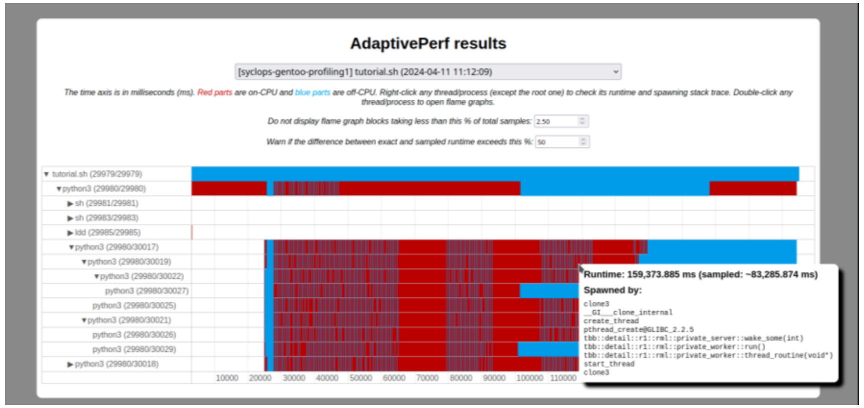
Maksymilian Graczyk (CERN, IT-GOV-INN)

What is AdaptivePerf?

NB: AdaptivePerf is **not** a continuous profiler! It profiles single commands and is not meant for 24/7 monitoring.

The main target audience is SW and HW developers optimising their software and/or hardware, also as part of software-hardware co-design for specific applications.

- **Open-source code profiler** for Linux, based on "perf" with custom patches and developed in the context of the SYCLOPS project
- Samples **both on-CPU and off-CPU** activity
- **Traces every spawned thread and process**
- **Minimises risk of broken profiled stacks** for programs compiled with frame pointers by detecting inappropriate kernel and CPU configurations automatically
- Produces **interactive flame graphs and charts** viewable in a web browser
- Main functionality designed with **hardware portability** in mind (tested on x86-64, RISC-V in progress, arm64 planned)
- Supports custom sampling-based "perf" events for **profiling interactions with hardware**
- **Allows TCP streaming** of profiling data to a separate machine for real-time processing



AdaptivePerf results

[syclops-gentoo-profiling1] tutorial.sh (2024-04-11 11:12:09)

The time axis is in milliseconds (ms). Red parts are on-CPU and blue parts are off-CPU. Right-click any thread/process (except the root one) to check its runtime and spawning stack trace. Double-click any thread/process to open flame graphs.

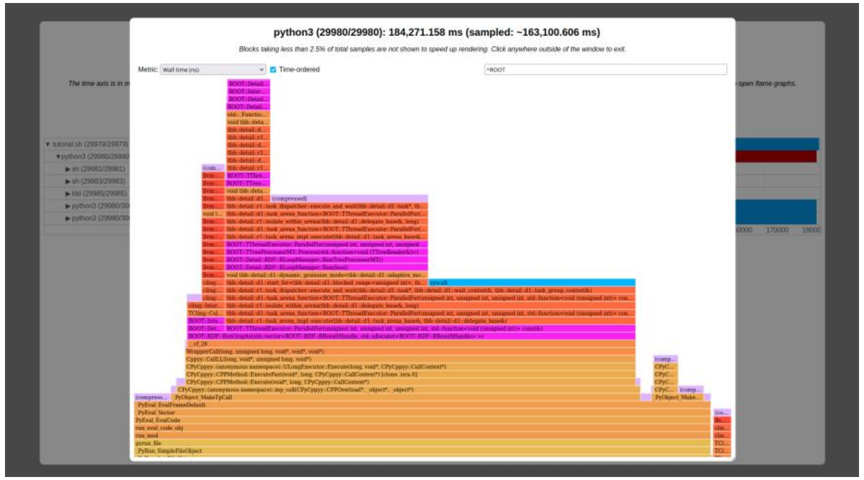
Do not display flame graph blocks taking less than this % of total samples: 2.50

Warn if the difference between exact and sampled runtime exceeds this %: 50

Runtime: 159,373.885 ms (sampled: ~83,285.874 ms)

Spawned by:

```
clone3
__GI___clone_internal
create_thread
pthread_create@GLIBC_2.2.5
tbb::detail::r1::rml::private_server::wake_some(int)
tbb::detail::r1::rml::private_worker::run()
tbb::detail::r1::rml::private_worker::thread_routine(void*)
start_thread
clone3
```



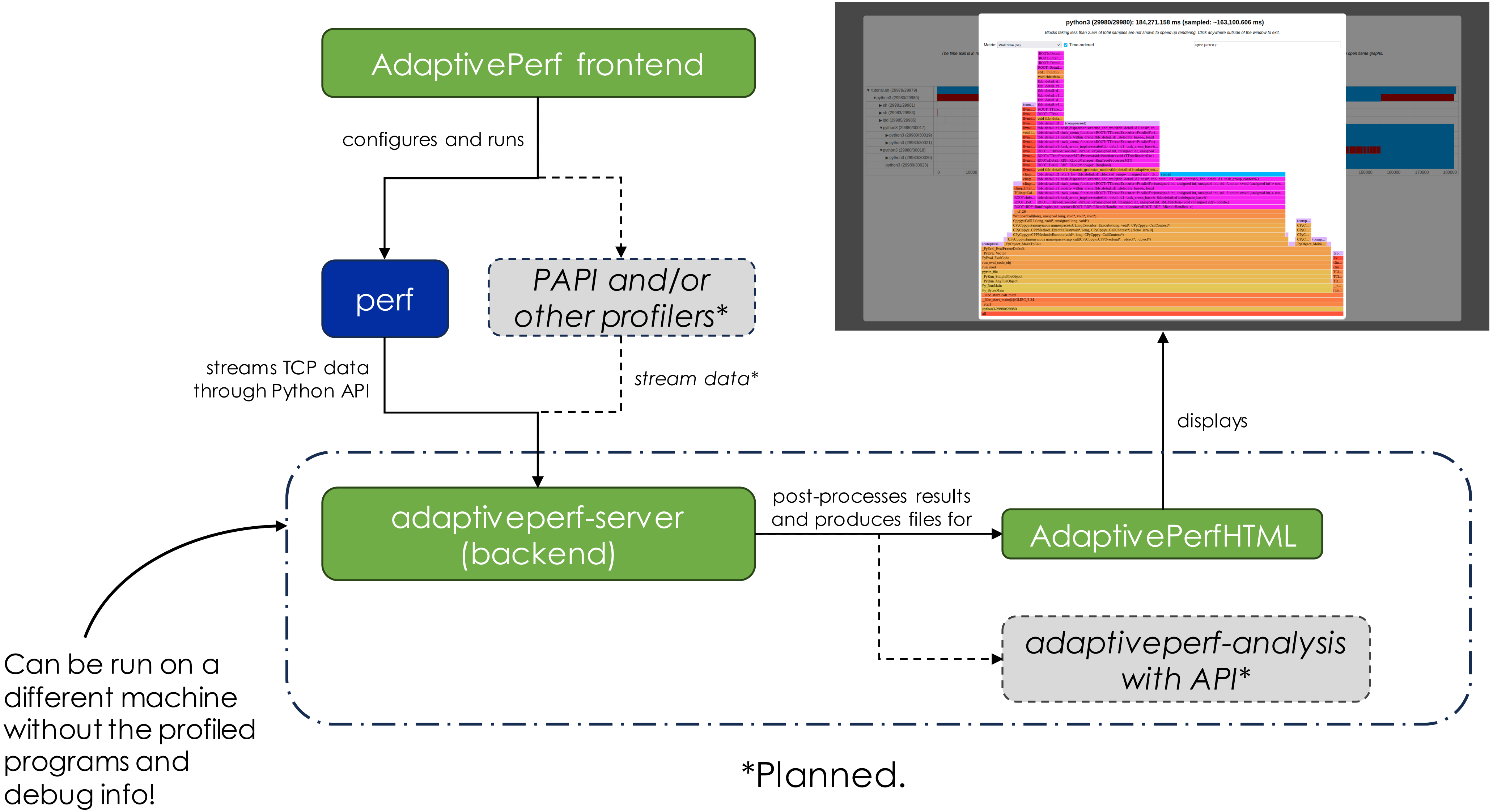
python3 (29980/29980): 184,271.158 ms (sampled: ~163,100.606 ms)

Blocks taking less than 2.5% of total samples are not shown to speed up rendering. Click anywhere outside of the window to exit.

Metric: Wall time (ns) ☑ Time-ordered

^ROOT

How does AdaptivePerf work under the hood?


AdaptivePerf frontend
configures and runs
perf
PAPI and/or other profilers*
streams TCP data through Python API
stream data*
adaptiveperf-server (backend)
post-processes results and produces files for
AdaptivePerfHTML
adaptiveperf-analysis with API*
displays
Can be run on a different machine without the profiled programs and debug info!

python3 (29980/29980): 184,271.158 ms (sampled: ~163,100.606 ms)

Blocks taking less than 2.5% of total samples are not shown to speed up rendering. Click anywhere outside of the window to exit.

Metric: Wall time (ns) ☑ Time-ordered ^(tbb|ROOT)::

*Planned.

What is SYCLOPS?

- An EU-funded project about hardware acceleration with open standards using SYCL and RISC-V
- Website: https://www.syclops.org
- CERN project tasks:
  1. Implementing SYCL support in ROOT and cling + demonstrating it on a Lorentz vector calculation example.
  2. Benchmarking and profiling + integration testing of all use cases envisaged in SYCLOPS (ROOT, genomics analysis, and autonomous systems).
- AdaptivePerf is part of task 2, but its applications extend beyond SYCLOPS!

Where outside of SYCLOPS can AdaptivePerf be potentially used?

- Profiling software used for online and offline computing at CERN and other physics experiments, e.g. Madgraph5 and Geant4
- Software-hardware co-design, e.g. in heterogeneous computing and development of triggering and DAQ systems at the LHC experiments
- And more!

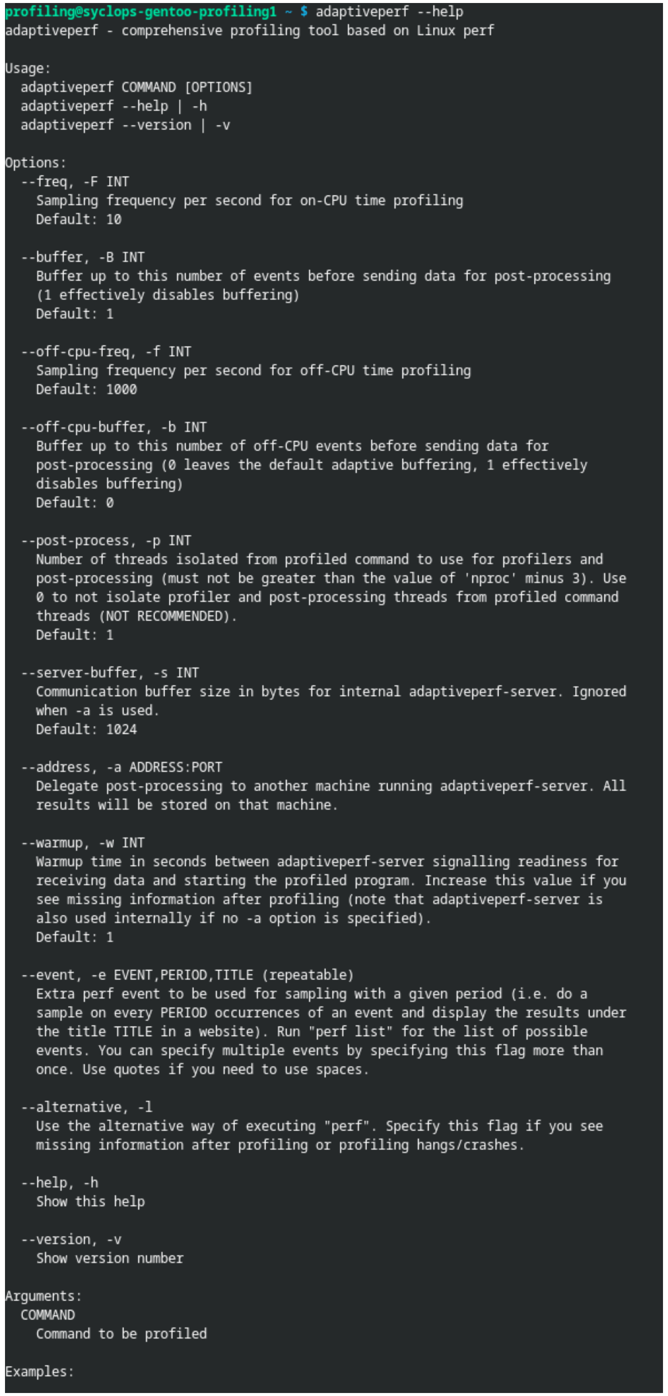
How to download AdaptivePerf?

- It's open-source and you can get it for free from our GitHub: https://github.com/AdaptivePerf.
- AdaptivePerf is available as a dev version. Feedback and feature requests are welcome.
- There are 3 parts:
  - AdaptivePerf: the main program which is the command-line profiling tool (frontend) and server (backend), licensed under GNU GPL v2 **only**.
  - AdaptivePerfHTML: the web server for displaying profiling results as an interactive website, licensed under GNU GPL v3.
  - Linux: the Linux kernel source tree with patched "perf", stored temporarily on CERN GitLab and licensed on the same terms as the vanilla Linux kernel (**only installing “perf” is required, no kernel patching needed**).

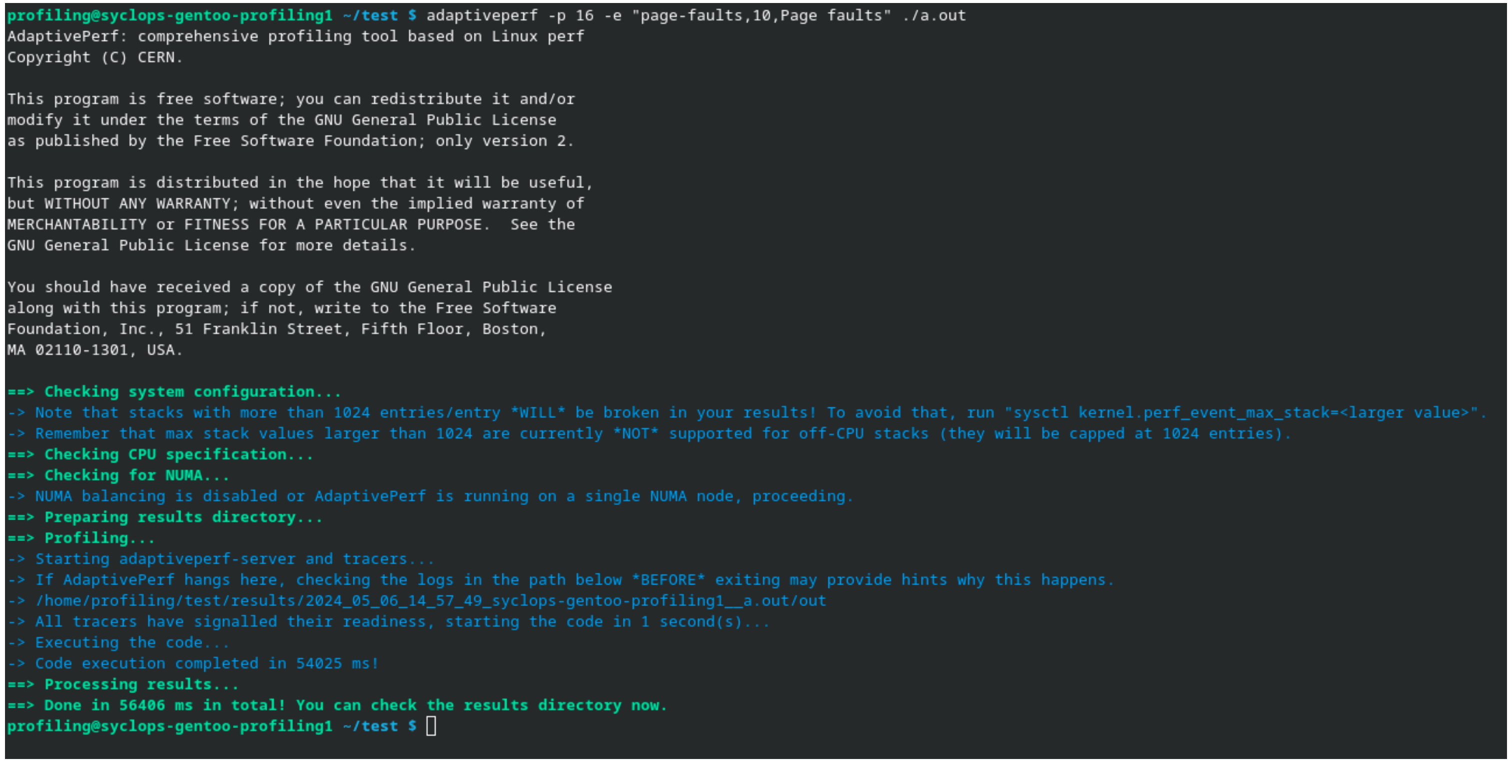
Quick start with AdaptivePerf

- Install AdaptivePerf and AdaptivePerfHTML according to the instructions on GitHub. Pay close attention there to the kernel settings and information about NUMA!
- Run `adaptiveperf "<command to be profiled>"` (quotes are important!) and wait until it finishes and produces the "results" directory.
- Set the `FLASK_PROFILING_STORAGE` environment variable to the "results" path.
- Run Flask (a Python web framework) and point it to AdaptivePerfHTML: `adaptiveperf.app:app`.
- Open the website in your web browser. Done!

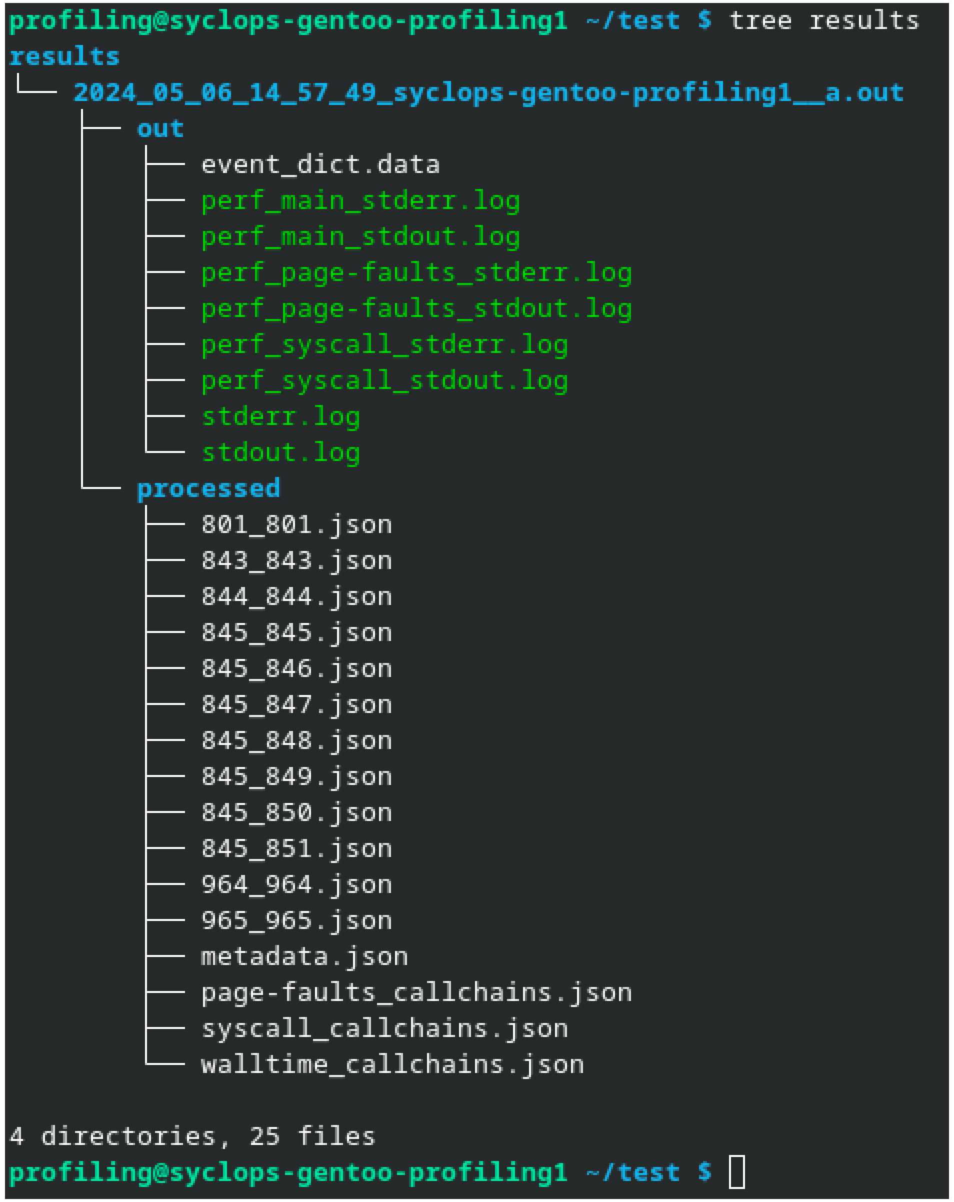
Live demo / Screenshots



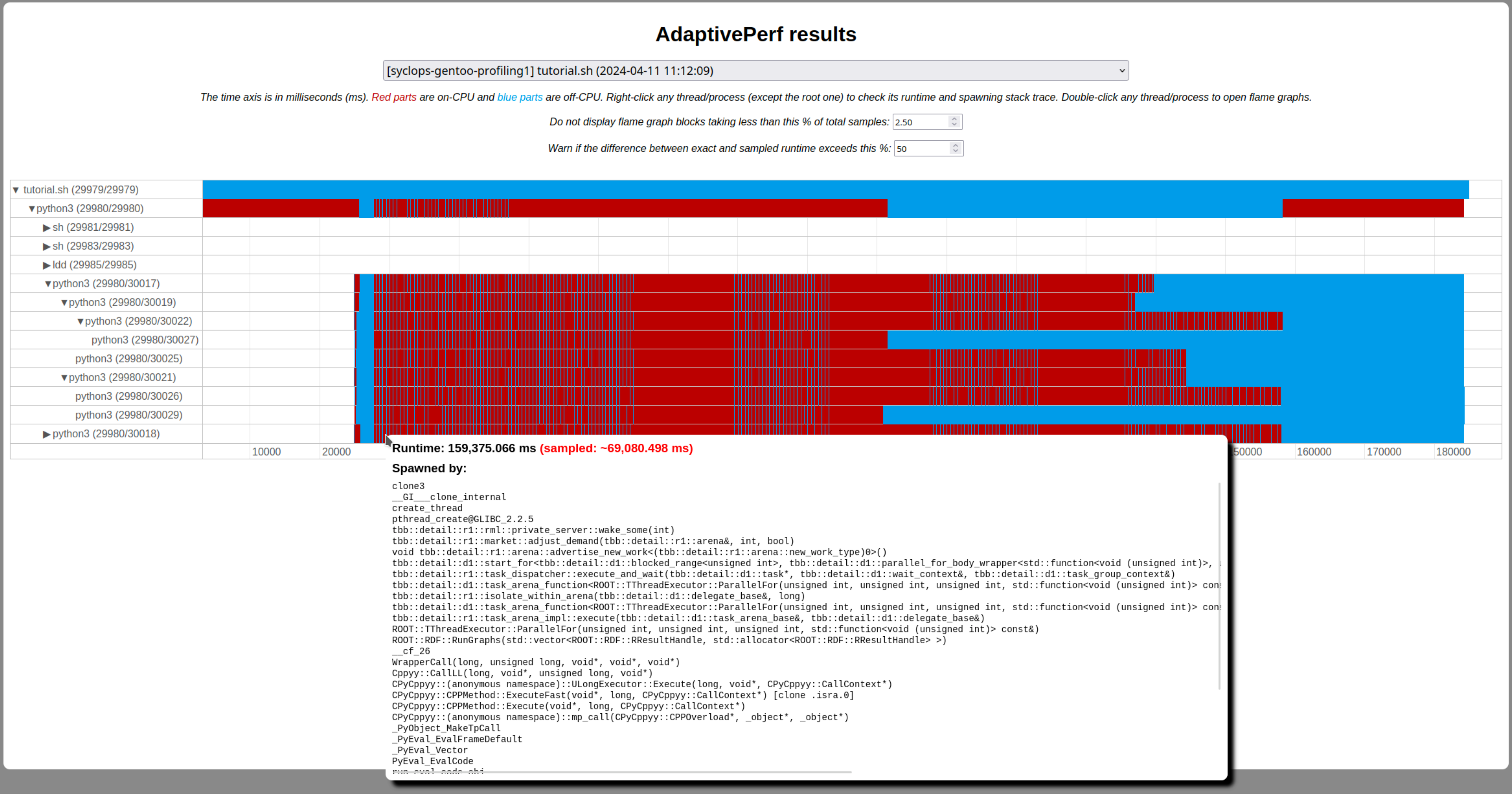
```
profiling@syclops-gentoo-profiling1 ~ $ adaptiveperf --help
adaptiveperf - comprehensive profiling tool based on Linux perf

Usage:
  adaptiveperf COMMAND [OPTIONS]
  adaptiveperf --help | -h
  adaptiveperf --version | -v

Options:
  --freq, -F INT
    Sampling frequency per second for on-CPU time profiling
    Default: 10

  --buffer, -B INT
    Buffer up to this number of events before sending data for post-processing
    (1 effectively disables buffering)
    Default: 1

  --off-cpu-freq, -f INT
    Sampling frequency per second for off-CPU time profiling
    Default: 1000

  --off-cpu-buffer, -b INT
    Buffer up to this number of off-CPU events before sending data for
    post-processing (0 leaves the default adaptive buffering, 1 effectively
    disables buffering)
    Default: 0

  --post-process, -p INT
    Number of threads isolated from profiled command to use for profilers and
    post-processing (must not be greater than the value of 'nproc' minus 3). Use
    0 to not isolate profiler and post-processing threads from profiled command
    threads (NOT RECOMMENDED).
    Default: 1

  --server-buffer, -s INT
    Communication buffer size in bytes for internal adaptiveperf-server. Ignored
    when -a is used.
    Default: 1024

  --address, -a ADDRESS:PORT
    Delegate post-processing to another machine running adaptiveperf-server. All
    results will be stored on that machine.

  --warmup, -w INT
    Warmup time in seconds between adaptiveperf-server signalling readiness for
    receiving data and starting the profiled program. Increase this value if you
    see missing information after profiling (note that adaptiveperf-server is
    also used internally if no -a option is specified).
    Default: 1

  --event, -e EVENT,PERIOD,TITLE (repeatable)
    Extra perf event to be used for sampling with a given period (i.e. do a
    sample on every PERIOD occurrences of an event and display the results under
    the title TITLE in a website). Run "perf list" for the list of possible
    events. You can specify multiple events by specifying this flag more than
    once. Use quotes if you need to use spaces.

  --alternative, -l
    Use the alternative way of executing "perf". Specify this flag if you see
    missing information after profiling or profiling hangs/crashes.

  --help, -h
    Show this help

  --version, -v
    Show version number

Arguments:
  COMMAND
    Command to be profiled

Examples:
```



```
profiling@syclops-gentoo-profiling1 ~/test $ adaptiveperf -p 16 -e "page-faults,10,Page faults" ./a.out
AdaptivePerf: comprehensive profiling tool based on Linux perf
Copyright (C) CERN.

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; only version 2.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
MA 02110-1301, USA.

==> Checking system configuration...
-> Note that stacks with more than 1024 entries/entry *WILL* be broken in your results! To avoid that, run "sysctl kernel.perf_event_max_stack=<larger value>".
-> Remember that max stack values larger than 1024 are currently *NOT* supported for off-CPU stacks (they will be capped at 1024 entries).
==> Checking CPU specification...
==> Checking for NUMA...
-> NUMA balancing is disabled or AdaptivePerf is running on a single NUMA node, proceeding.
==> Preparing results directory...
==> Profiling...
-> Starting adaptiveperf-server and tracers...
-> If AdaptivePerf hangs here, checking the logs in the path below *BEFORE* exiting may provide hints why this happens.
-> /home/profiling/test/results/2024_05_06_14_57_49_syclops-gentoo-profiling1__a.out/out
-> All tracers have signalled their readiness, starting the code in 1 second(s)...
-> Executing the code...
-> Code execution completed in 54025 ms!
==> Processing results...
==> Done in 56406 ms in total! You can check the results directory now.
profiling@syclops-gentoo-profiling1 ~/test $
```



```
profiling@syclops-gentoo-profiling1 ~/test $ tree results
results
└── 2024_05_06_14_57_49_syclops-gentoo-profiling1__a.out
    ├── out
    │   ├── event_dict.data
    │   ├── perf_main_stderr.log
    │   ├── perf_main_stdout.log
    │   ├── perf_page-faults_stderr.log
    │   ├── perf_page-faults_stdout.log
    │   ├── perf_syscall_stderr.log
    │   ├── perf_syscall_stdout.log
    │   ├── stderr.log
    │   └── stdout.log
    └── processed
        ├── 801_801.json
        ├── 843_843.json
        ├── 844_844.json
        ├── 845_845.json
        ├── 845_846.json
        ├── 845_847.json
        ├── 845_848.json
        ├── 845_849.json
        ├── 845_850.json
        ├── 845_851.json
        ├── 964_964.json
        ├── 965_965.json
        ├── metadata.json
        ├── page-faults_callchains.json
        ├── syscall_callchains.json
        └── walltime_callchains.json

4 directories, 25 files
profiling@syclops-gentoo-profiling1 ~/test $
```



AdaptivePerf results

[syclops-gentoo-profiling1] tutorial.sh (2024-04-11 11:12:09)

The time axis is in milliseconds (ms). Red parts are on-CPU and blue parts are off-CPU. Right-click any thread/process (except the root one) to check its runtime and spawning stack trace. Double-click any thread/process to open flame graphs.

Do not display flame graph blocks taking less than this % of total samples: 2.50

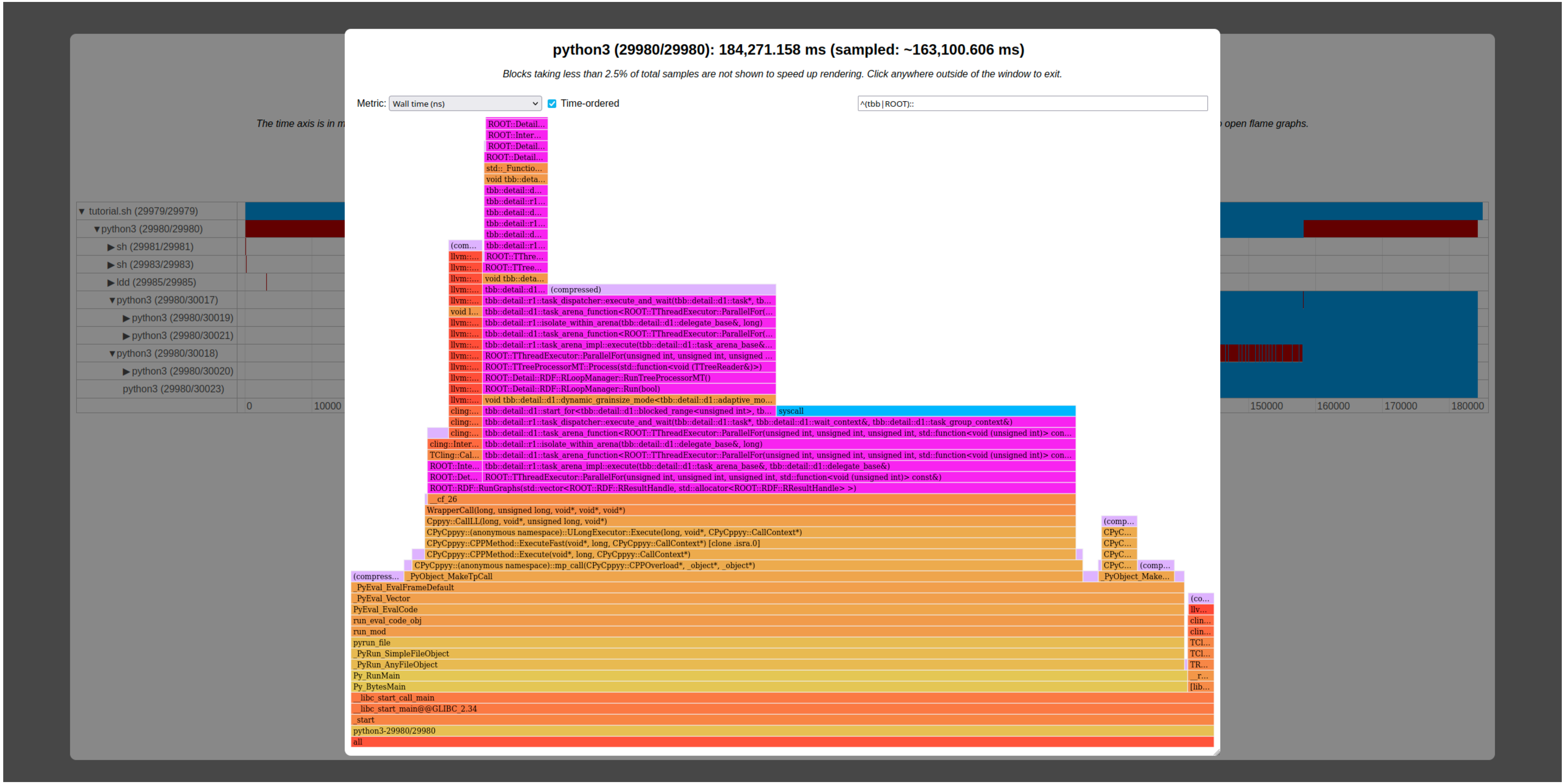
Warn if the difference between exact and sampled runtime exceeds this %: 50

```
▼ tutorial.sh (29979/29979)
  ▼python3 (29980/29980)
    ▶ sh (29981/29981)
    ▶ sh (29983/29983)
    ▶ ldd (29985/29985)
    ▼python3 (29980/30017)
      ▼python3 (29980/30019)
        ▼python3 (29980/30022)
          python3 (29980/30027)
        python3 (29980/30025)
      ▼python3 (29980/30021)
        python3 (29980/30026)
        python3 (29980/30029)
    ▶ python3 (29980/30018)
```

Runtime: 159,375.066 ms (sampled: ~69,080.498 ms)

Spawned by:

```
clone3
__GI___clone_internal
create_thread
pthread_create@GLIBC_2.2.5
tbb::detail::r1::rml::private_server::wake_some(int)
tbb::detail::r1::market::adjust_demand(tbb::detail::r1::arena&, int, bool)
void tbb::detail::r1::arena::advertise_new_work<(tbb::detail::r1::arena::new_work_type)0>()
tbb::detail::d1::start_for<tbb::detail::d1::blocked_range<unsigned int>, tbb::detail::d1::parallel_for_body_wrapper<std::function<void (unsigned int)>,
tbb::detail::r1::task_dispatcher::execute_and_wait(tbb::detail::d1::task*, tbb::detail::d1::wait_context&, tbb::detail::d1::task_group_context&)
tbb::detail::d1::task_arena_function<ROOT::TThreadExecutor::ParallelFor(unsigned int, unsigned int, unsigned int, std::function<void (unsigned int)> con
tbb::detail::r1::isolate_within_arena(tbb::detail::d1::delegate_base&, long)
tbb::detail::d1::task_arena_function<ROOT::TThreadExecutor::ParallelFor(unsigned int, unsigned int, unsigned int, std::function<void (unsigned int)> con
tbb::detail::r1::task_arena_impl::execute(tbb::detail::d1::task_arena_base&, tbb::detail::d1::delegate_base&)
ROOT::TThreadExecutor::ParallelFor(unsigned int, unsigned int, unsigned int, std::function<void (unsigned int)> const&)
ROOT::RDF::RunGraphs(std::vector<ROOT::RDF::RResultHandle, std::allocator<ROOT::RDF::RResultHandle> >)
__cf_26
WrapperCall(long, unsigned long, void*, void*, void*)
Cppyy::CallLL(long, void*, unsigned long, void*)
CPyCppyy::(anonymous namespace)::ULongExecutor::Execute(long, void*, CPyCppyy::CallContext*)
CPyCppyy::CPPMethod::ExecuteFast(void*, long, CPyCppyy::CallContext*) [clone .isra.0]
CPyCppyy::CPPMethod::Execute(void*, long, CPyCppyy::CallContext*)
CPyCppyy::(anonymous namespace)::mp_call(CPyCppyy::CPPOverload*, _object*, _object*)
_PyObject_MakeTpCall
_PyEval_EvalFrameDefault
_PyEval_Vector
PyEval_EvalCode
run_eval_code_obj
```

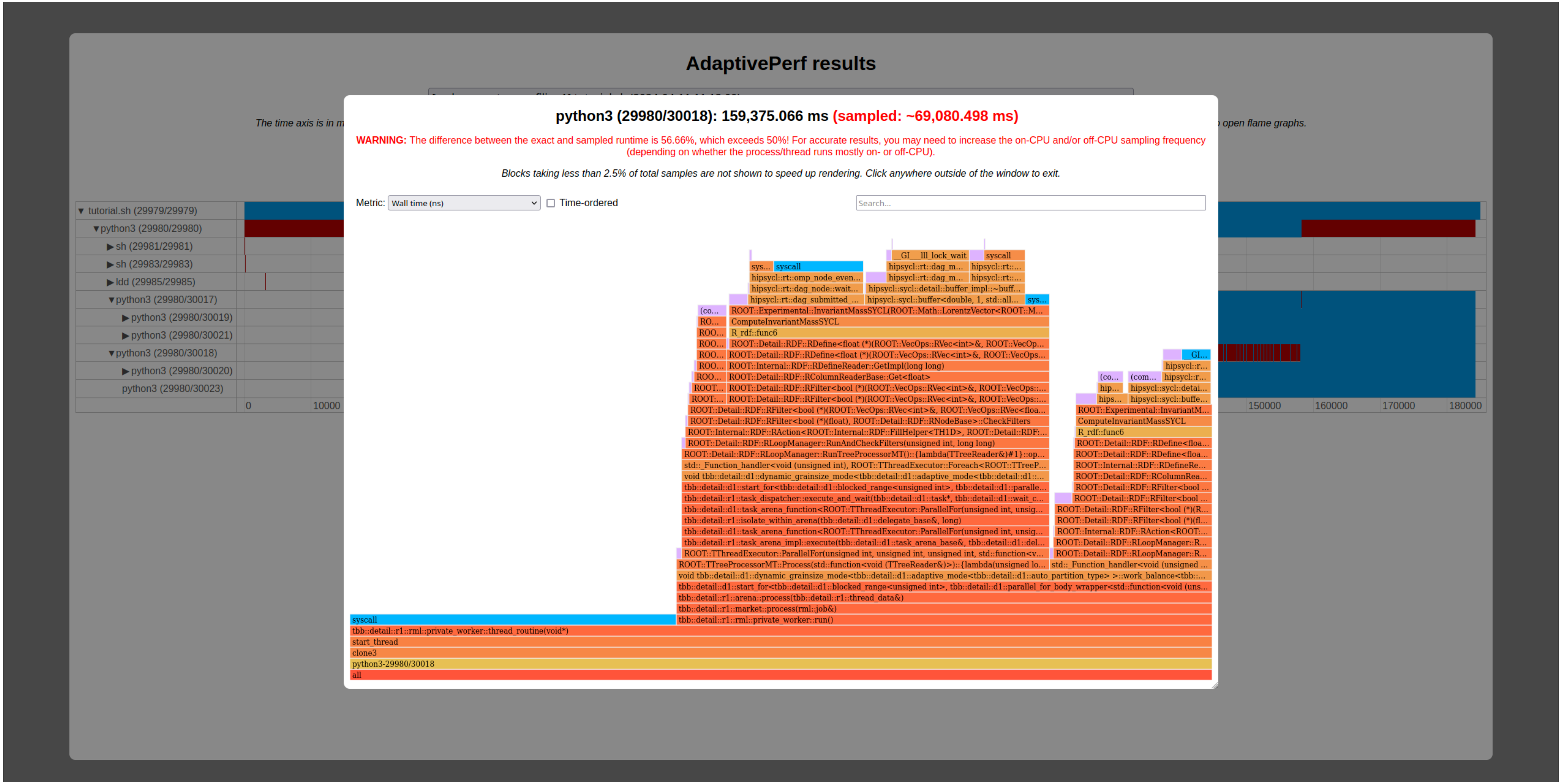


python3 (29980/29980): 184,271.158 ms (sampled: ~163,100.606 ms)

Blocks taking less than 2.5% of total samples are not shown to speed up rendering. Click anywhere outside of the window to exit.

Metric: Wall time (ns) ☑ Time-ordered ^(tbb|ROOT)::

The time axis is in m
open flame graphs.
▼ tutorial.sh (29979/29979)
▼ python3 (29980/29980)
▶ sh (29981/29981)
▶ sh (29983/29983)
▶ ldd (29985/29985)
▼ python3 (29980/30017)
▶ python3 (29980/30019)
▶ python3 (29980/30021)
▼ python3 (29980/30018)
▶ python3 (29980/30020)
python3 (29980/30023)
0
10000
150000
160000
170000
180000
ROOT::Detail...
ROOT::Inter...
ROOT::Detail...
ROOT::Detail...
std::_Functio...
void tbb::deta...
tbb::detail::d...
tbb::detail::r1...
tbb::detail::d...
tbb::detail::r1...
tbb::detail::d...
(com...
tbb::detail::r1...
llvm::...
ROOT::TThre...
ROOT::TTree...
void tbb::deta...
tbb::detail::d1...
(compressed)
tbb::detail::r1::task_dispatcher::execute_and_wait(tbb::detail::d1::task*, tb...
void l...
tbb::detail::d1::task_arena_function<ROOT::TThreadExecutor::ParallelFor(...
tbb::detail::r1::isolate_within_arena(tbb::detail::d1::delegate_base&, long)
tbb::detail::d1::task_arena_function<ROOT::TThreadExecutor::ParallelFor(...
tbb::detail::r1::task_arena_impl::execute(tbb::detail::d1::task_arena_base&...
ROOT::TThreadExecutor::ParallelFor(unsigned int, unsigned int, unsigned ...
ROOT::TTreeProcessorMT::Process(std::function<void (TTreeReader&)>)
ROOT::Detail::RDF::RLoopManager::RunTreeProcessorMT()
ROOT::Detail::RDF::RLoopManager::Run(bool)
void tbb::detail::d1::dynamic_grainsize_mode<tbb::detail::d1::adaptive_mo...
cling::...
tbb::detail::d1::start_for<tbb::detail::d1::blocked_range<unsigned int>, tb...
syscall
tbb::detail::r1::task_dispatcher::execute_and_wait(tbb::detail::d1::task*, tbb::detail::d1::wait_context&, tbb::detail::d1::task_group_context&)
tbb::detail::d1::task_arena_function<ROOT::TThreadExecutor::ParallelFor(unsigned int, unsigned int, unsigned int, std::function<void (unsigned int)> con...
cling::Inter...
tbb::detail::r1::isolate_within_arena(tbb::detail::d1::delegate_base&, long)
TCling::Cal...
tbb::detail::d1::task_arena_function<ROOT::TThreadExecutor::ParallelFor(unsigned int, unsigned int, unsigned int, std::function<void (unsigned int)> con...
ROOT::Inte...
tbb::detail::r1::task_arena_impl::execute(tbb::detail::d1::task_arena_base&, tbb::detail::d1::delegate_base&)
ROOT::Det...
ROOT::TThreadExecutor::ParallelFor(unsigned int, unsigned int, unsigned int, std::function<void (unsigned int)> const&)
ROOT::RDF::RunGraphs(std::vector<ROOT::RDF::RResultHandle, std::allocator<ROOT::RDF::RResultHandle> >)
__cf_26
WrapperCall(long, unsigned long, void*, void*, void*)
Cppyy::CallLL(long, void*, unsigned long, void*)
CPyCppyy::(anonymous namespace)::ULongExecutor::Execute(long, void*, CPyCppyy::CallContext*)
CPyCppyy::CPPMethod::ExecuteFast(void*, long, CPyCppyy::CallContext*) [clone .isra.0]
CPyCppyy::CPPMethod::Execute(void*, long, CPyCppyy::CallContext*)
CPyCppyy::(anonymous namespace)::mp_call(CPyCppyy::CPPOverload*, _object*, _object*)
(compress...
_PyObject_MakeTpCall
(comp...
CPyC...
(comp...
_PyObject_Make...
_PyEval_EvalFrameDefault
_PyEval_Vector
(co...
PyEval_EvalCode
llv...
run_eval_code_obj
clin...
run_mod
clin...
pyrun_file
TCl...
_PyRun_SimpleFileObject
TCl...
_PyRun_AnyFileObject
TR...
Py_RunMain
_r...
Py_BytesMain
[lib...
__libc_start_call_main
__libc_start_main@@GLIBC_2.34
_start
python3-29980/29980
all



AdaptivePerf results

The time axis is in m… …open flame graphs.

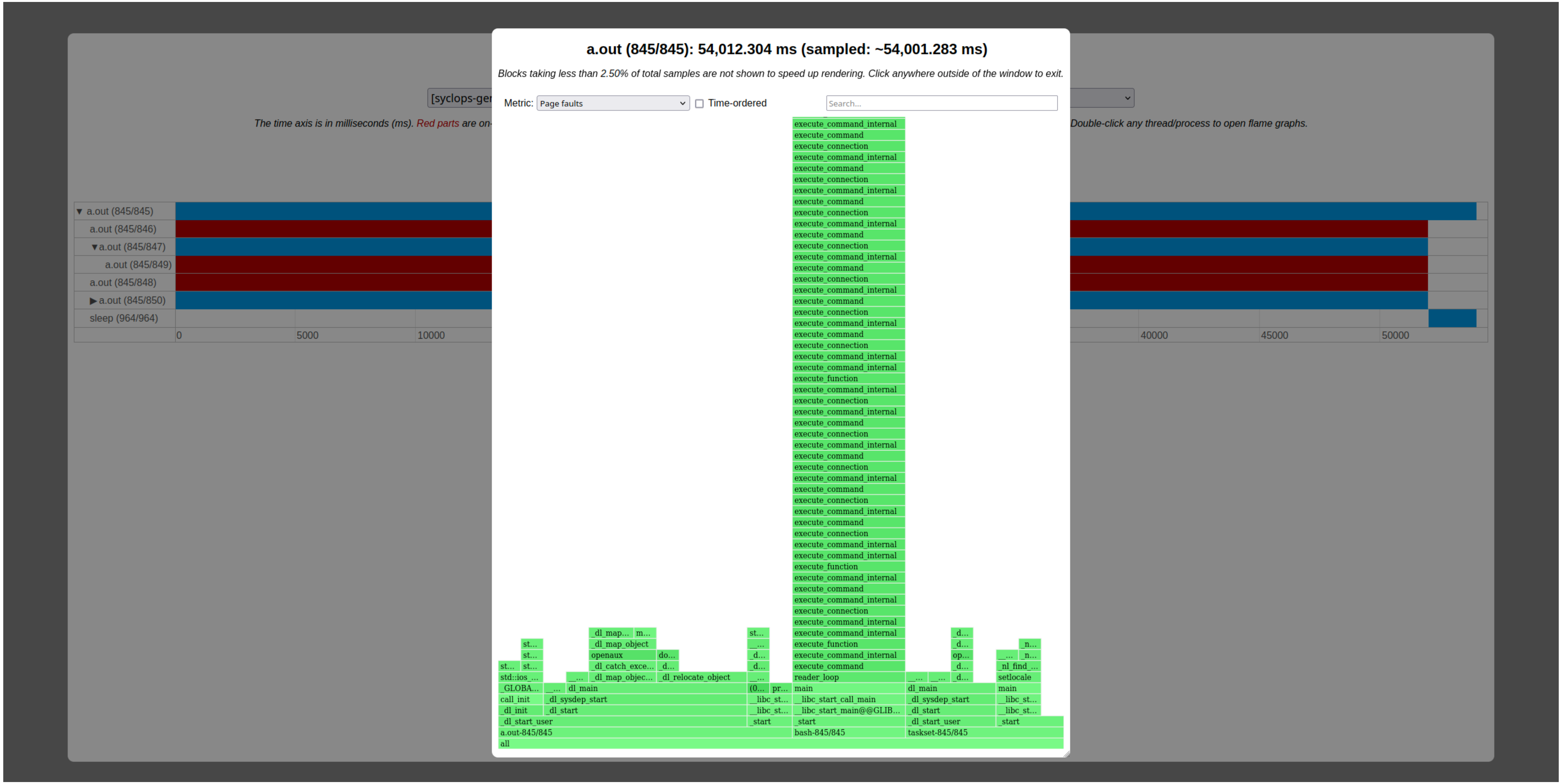
python3 (29980/30018): 159,375.066 ms (sampled: ~69,080.498 ms)

WARNING: The difference between the exact and sampled runtime is 56.66%, which exceeds 50%! For accurate results, you may need to increase the on-CPU and/or off-CPU sampling frequency (depending on whether the process/thread runs mostly on- or off-CPU).

Blocks taking less than 2.5% of total samples are not shown to speed up rendering. Click anywhere outside of the window to exit.

Metric: Wall time (ns) ☐ Time-ordered

Search...
▼ tutorial.sh (29979/29979)
▼python3 (29980/29980)
▶ sh (29981/29981)
▶ sh (29983/29983)
▶ ldd (29985/29985)
▼python3 (29980/30017)
▶ python3 (29980/30019)
▶ python3 (29980/30021)
▼python3 (29980/30018)
▶ python3 (29980/30020)
python3 (29980/30023)
0
10000
150000
160000
170000
180000
_GI__lll_lock_wait
syscall
sys...
syscall
hipsycl::rt::dag_m...
hipsycl::rt::...
hipsycl::rt::omp_node_even...
hipsycl::rt::dag_node::wait...
hipsycl::sycl::detail::buffer_impl::~buff...
hipsycl::rt::dag_submitted_...
hipsycl::sycl::buffer&lt;double, 1, std::all...
sys...
ROOT::Experimental::InvariantMassSYCL(ROOT::Math::LorentzVector&lt;ROOT::M...
ComputeInvariantMassSYCL
R_rdf::func6
ROOT::Detail::RDF::RDefine&lt;float (*)(ROOT::VecOps::RVec&lt;int&gt;&amp;, ROOT::VecOp...
ROOT::Internal::RDF::RDefineReader::GetImpl(long long)
ROOT::Detail::RDF::RColumnReaderBase::Get&lt;float&gt;
ROOT::Detail::RDF::RFilter&lt;bool (*)(ROOT::VecOps::RVec&lt;int&gt;&amp;, ROOT::VecOps:...
ROOT::Detail::RDF::RFilter&lt;bool (*)(float), ROOT::Detail::RDF::RNodeBase&gt;::CheckFilters
ROOT::Internal::RDF::RAction&lt;ROOT::Internal::RDF::FillHelper&lt;TH1D&gt;, ROOT::Detail::RDF:...
ROOT::Detail::RDF::RLoopManager::RunAndCheckFilters(unsigned int, long long)
ROOT::Detail::RDF::RLoopManager::RunTreeProcessorMT()::{lambda(TTreeReader&amp;)#1}::op...
std::_Function_handler&lt;void (unsigned int), ROOT::TThreadExecutor::Foreach&lt;ROOT::TTreeP...
void tbb::detail::d1::dynamic_grainsize_mode&lt;tbb::detail::d1::adaptive_mode&lt;tbb::detail::d1::...
tbb::detail::d1::start_for&lt;tbb::detail::d1::blocked_range&lt;unsigned int&gt;, tbb::detail::d1::paralle...
tbb::detail::r1::task_dispatcher::execute_and_wait(tbb::detail::d1::task*, tbb::detail::d1::wait_c...
tbb::detail::d1::task_arena_function&lt;ROOT::TThreadExecutor::ParallelFor(unsigned int, unsig...
tbb::detail::r1::isolate_within_arena(tbb::detail::d1::delegate_base&amp;, long)
tbb::detail::r1::task_arena_impl::execute(tbb::detail::d1::task_arena_base&amp;, tbb::detail::d1::del...
ROOT::TThreadExecutor::ParallelFor(unsigned int, unsigned int, unsigned int, std::function&lt;v...
ROOT::TTreeProcessorMT::Process(std::function&lt;void (TTreeReader&amp;)&gt;)::{lambda(unsigned lo...
std::_Function_handler&lt;void (unsigned ...
void tbb::detail::d1::dynamic_grainsize_mode&lt;tbb::detail::d1::adaptive_mode&lt;tbb::detail::d1::auto_partition_type&gt; &gt;::work_balance&lt;tbb::...
tbb::detail::d1::start_for&lt;tbb::detail::d1::blocked_range&lt;unsigned int&gt;, tbb::detail::d1::parallel_for_body_wrapper&lt;std::function&lt;void (uns...
tbb::detail::r1::arena::process(tbb::detail::r1::thread_data&amp;)
tbb::detail::r1::market::process(rml::job&amp;)
syscall
tbb::detail::r1::rml::private_worker::run()
tbb::detail::r1::rml::private_worker::thread_routine(void*)
start_thread
clone3
python3-29980/30018
all



a.out (845/845): 54,012.304 ms (sampled: ~54,001.283 ms)

Blocks taking less than 2.50% of total samples are not shown to speed up rendering. Click anywhere outside of the window to exit.

Metric: Page faults ☐ Time-ordered

The time axis is in milliseconds (ms). Red parts are on- … *Double-click any thread/process to open flame graphs.*

How does AdaptivePerf compare to other similar and maintained profilers?

	Hardware-vendor-portable	Profiles software-hardware interaction*	Low profiling overhead	Open-source	Off-CPU profiling	Heterogeneous architecture support
AdaptivePerf	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Planned!**
Original "perf"	Yes	Yes	Yes	Yes	Limited	No
Intel VTune Profiler	No	Yes	Yes	No	Yes	Intel GPUs/FPGAs only
AMD μProf	No	Yes	Yes	No	Yes	AMD GPUs only
valgrind	Yes	No	No	Yes	No	No
gprof	Yes	No	Needs CI**	Yes	No	No
gperftools	Yes	No	Needs CI**	Yes	No	No
NVIDIA profilers	No	Yes	No	No	Yes	NVIDIA GPUs only

*If supported by a user's hardware architecture.

**Code instrumentation other than not omitting frame pointers.

Future plans

- Profiling heterogeneous architectures and non-CPU devices in a maximally open-source way
  - One idea of doing this is through PAPI: https://github.com/icl-utk-edu/papi.
- Applying AdaptivePerf to cache-aware roofline modelling and potentially RISC-V core customisation by collaborating with some of our SYCLOPS partners: INESC-ID, Codasip, EURECOM
- Expanding the analysis functionality by making a separate library with the API and adding the plugin API to AdaptivePerfHTML
- Setting up automated tests (already in progress)

Future plans

- Adding profiling on a lower level and with more debug info, e.g. showing line numbers, going down to LLVM IR / MLIR / assembly etc.
- Downloading debug info for a given process automatically if not present, e.g. through debuginfod (a server providing debugging information, there are public ones available)
- Matching non-sampling-based metrics from “perf” and/or other programs (e.g. power consumption) to code segments
  - An openlab summer student is coming to CERN on 1 July to work on this.

Future plans

- Decreasing profiling overhead even more
  - For example, by replacing "perf"'s Python API with its C/C++/Rust/... equivalent. This may require another set of "perf" patches, as "perf" supports only Python and Perl out-of-the-box.
- Removing or weakening the frame pointer compilation requirement
  - For example, by DWARF processing whenever frame pointers cannot be used, see: https://www.polarsignals.com/blog/posts/2022/11/29/profiling-without-frame-pointers (this is more compact than what "perf" currently does).
  - Full removal may be unnecessary, see: https://brendangregg.com/blog/2024-03-17/the-return-of-the-frame-pointers.html.
- All other suggestions are welcome!

Thank you!

Any questions or comments?