IPCC ROOT
Princeton/Intel Parallel Computing Center

Showcase Presentation

PI Peter Elmer

Vassil Vassilev, Oksana Shadura, Yuka Takahashi

08.11.2018

PRINCETON UNIVERSITY

Outline

- IPCC-ROOT. Plan of work. Goals
- Code modernization:
    - Enable Continuous Performance Integration
    - Modernize ROOT's Math packages by integrating clad
    - Optimize ROOT's I/O and dictionary format employing C++ Modules
    - Optimize ROOT's reflection layer
  - Future directions
  - Other activities & Outreach

IPCC-ROOT

- ROOT is in the core of HEP experiments (including LHC's ALICE, ATLAS, CMS, LHCb) and around 1EB of data is stored in ROOT files. Even a small improvement in ROOT could have significant impact on the HEP community
- Princeton/Intel Parallel Computing Center to modernize ROOT funded via Intel's Parallel Computing Center (IPCC) program
- Started in 2017 in coordination with CERN OpenLab and the ROOT Team
- 1 full time (Vassil) engineer employed for 1 (+1) year, located at CERN, member of the ROOT team, plus some NSF-funded DIANA/HEP collaboration (O.Shadura, Y.Takahashi)

Work plan 2018

Component in ROOT	Deliverable	Success Criteria	Period
Infrastructure	**Enable Continuous Performance Integration:** In Y1 we implemented various microbenchmarks which test code scalability (esp with respect to threading and vectorisation). We would like to continue extending them and running them on a nightly basis. Automatizing the process would allow us to find performance regressions. Another direct benefit would be that we can provide more detailed comparisons between compilers, compiler versions, compiler switches, libraries, operating systems and various Intel hardware. Currently the process is very laborious and takes a lot of developer's time which can be replaced by this automatic infrastructure making it a matter of setting up a configuration matrix.	Run ROOT's benchmarks nightly on Intel hardware	Q1

Work plan 2018

Component in ROOT	Deliverable	Success Criteria	Period
Math	**Modernize ROOT's Math packages by integrating clad:** Y1, Q4 delivers clad: a tool to speed up the production of derivatives. RooFit and TMVA are one of the major places where clad can be used. Currently, the only foreseen derivation backend is employing the numerical differentiation. Clad can be implemented as another backend which delivers derivatives.	Enable a clad-based derivative backend	Q2

Work plan 2018

Component in ROOT	Deliverable	Success Criteria	Period
I/O and Reflection	**Optimize ROOT's I/O and dictionary format employing C++ Modules**: ROOT's I/O and reflection layers performs an essential role in the overall performance of ROOT. Currently, ROOT uses its C++ interpreter, cling, to learn about memory layout and other important properties of C++ entities in order to perform correct and efficient on-disk serialization or deserialization. Cling, parses source code to understand the object layouts. In many cases the parsing slows down the overall system performance. We can reduce the amounts of parsing by introducing C++ modules. This in turn will reduce the locking times in the reflection layer, making ROOT more robust when used in multithreaded environments.	Enable C++ Modules as a reflection dictionary provider	Q3

Work plan 2018

Component in ROOT	Deliverable	Success Criteria	Period
I/O and Reflection	**Optimize ROOT's reflection layer:** In a few places ROOT asks for reflection information eagerly which causes the interpreter to activate locks and reduce the parallel execution. Instead, ROOT's reflection layer should request only the minimal amount of type information lazily. This in turn will reduce the locking times in the reflection layer, making ROOT more robust when used in multithreaded environments.	Reduce ROOT's locking times	Q4

Working Environment

Performance measurements are done on:

- [Vassil] Mac OS X, 2.5 GHz Intel Core i7, 16 GB
- [Yuka] Archlinux 4.18.16 GNU/Linux,Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz, 16 GB DDR4 , 1xSSD 512 GB
- [NUC] Ubuntu 18.04 , kernel 4.15.0-38-generic, i7-8809G Processor with Radeon™ RX Vega M GH graphics (8M Cache, up to 4.20 GHz), 2x16 GB DDR4 2666 , 1xSSD 512 GB (latest Intel NUC Hades Canyon)
- [Oksana] Ubuntu 18.04.1 LTS, Lenovo Thinkpad E470 i7-7500U NVIDIA GeForce 940MX, 16GB RAM, 256GB SSD
- [OpenLab] CentOS 7.3 kernel 3.10.0-514.26.2.el7.x86_64, Intel Xeon CPU E5-2683 v3 @ 2.00GHz, 14 core (dual socket system => 14x2x2 = up to 56 logical), 64 GB DDR4, 2xSSDs 240GB (latest Haswell)

Code Modernization in ROOT. Enable Continuous Performance Integration

Run ROOT's benchmarks nightly on Intel hardware

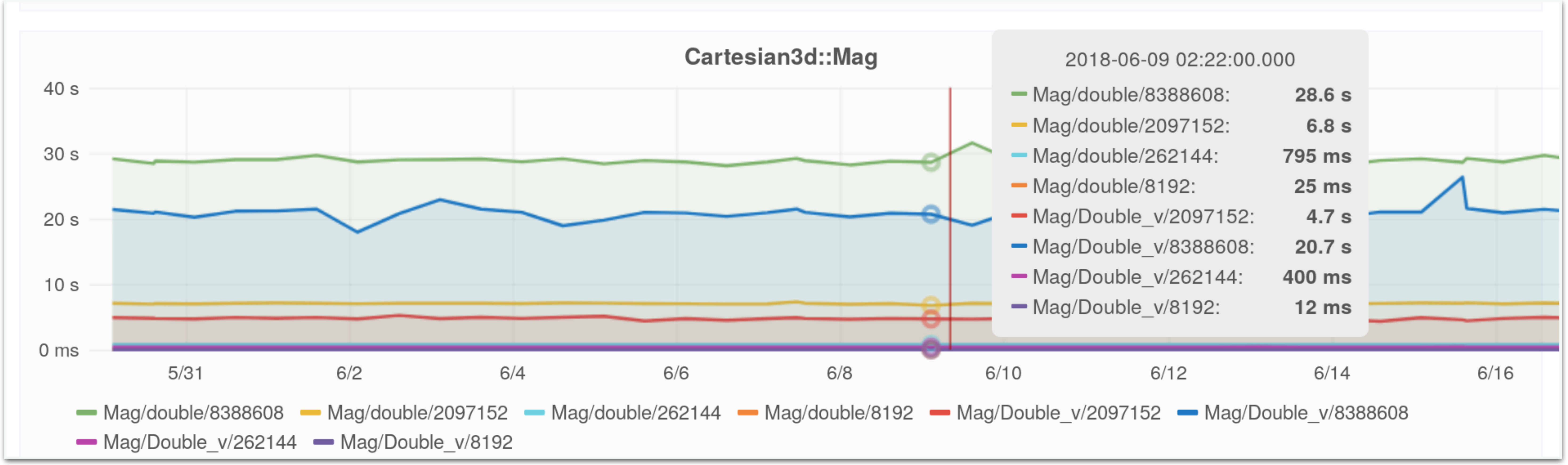
Completed Q1 Deliverable (available at https://rootbnch-grafana-test.cern.ch)

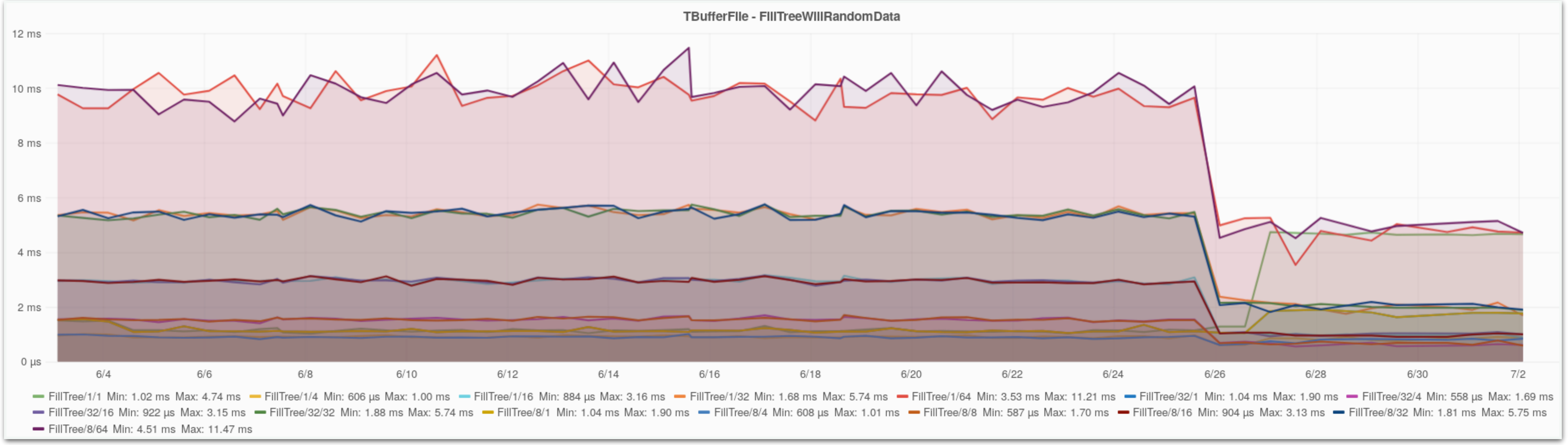
Continuous Performance Integration. Goals

- Observe performance improvements and guarantee their sustainability
- Monitor continuously the framework's performance
- Visualize performance regressions
- Support flexible and extensible benchmarks and metrics (such as cpu time, memory usage and on-disk size)
- Measurements done on [OpenLab]


Firefox File Edit View History Bookmarks Tools Window Help
100% Wed Nov 7 15:16
Grafana - ROOT Health Dashbo
https://rootbnch-grafana-test.cern.ch/dashboard/db/root-health-dashboard
67%
Search
ROOT Health Dashboard
Zoom Out
Last 30 days
Find dashboards by name
starred | tags
Hist benchmarks - Haswell
IO benchmarks - TBufferMerger
IO benchmarks - TBufferMerger - KNL
Interpreter benchmarks - cxxmodules - Memory - Haswell
Interpreter benchmarks - cxxmodules - Memory - SandyBridge
Interpreter benchmarks - cxxmodules - Haswell
Interpreter benchmarks - cxxmodules - KNL (Historical measurements))
Interpreter benchmarks - cxxmodules - Sandy Bridge
Python Interpreter benchmarks - Haswell
RDF Dashboard
ROOT Health Dashboard
Roofit benchmarks - Roofit Binned & Unbinned Benchmarks
ROOT health statistics - RSS (hsimple)
N/A
ROOT health statistics - RT (hsimple)
813 ms
ROOT health statistics - CPU time (hsimple)
450 µs
ROOT C++ modules health statistics - RSS (mean value of all benchmarks)
115.5 MiB
ROOT C++ modules health statistics - RT (mean value of all benchmarks)
708 ms

Continuous Performance Integration. Results





Continuous Performance Integration. Results

Continuous Performance Integration. Results

- The technology is **the** ROOT performance monitoring system (publicly accessible through ROOT's homepage, see "Development/Benchmarks" at https://root.cern)
- Verification of benchmarks now a required step for releases, see step 3 of https://root.cern/release-checklist
- Other projects (in particular Geant) start working on similar system using the same set of technologies

Continuous Performance Integration. Publications & Outreach

- Continuous Performance Benchmarking Framework for ROOT, Poster at CHEP, 9-13 July 2018, Sofia, Bulgaria
- Many well-received CERN-internal presentations

Continuous Performance Integration. Future Work

- Increase the micro benchmark coverage
- Track regressions and send alarms
- Automatically generate flame graphs
- Integrate it into the pull request development model of ROOT

Code Modernization in ROOT. Modernize ROOT's Math packages by integrating clad

Enable a clad-based derivative backend

Completed Q2 Deliverable (available in ROOT v6.14 and ROOT v6.16)

Automatic Differentiation in a Nutshell. Clad

Automatic differentiation is superior to the slow symbolic or often inaccurate numerical differentiation. It uses the fact that every computer program can be divided into a set of elementary operations (-,+,*,/) and functions (sin, cos, log, etc). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed. See more at the IPCC-ROOT Showcase Presentation in 2017.

Clad is a C/C++ to C/C++ language transformer implementing the chain rule from differential calculus. For example:

```
constexpr double MyPow(double x) { return x*x; }
```

↓

```
constexpr double MyPow_darg0(double x) { return (1. * x + x * 1.); }
```

Clad. Goals

- Improve numerical stability and correctness
- Replace iterative algorithms computing gradients with a single function call (of a interpreter-generated routine)
- Provide an alternative way of gradient computations in ROOT's fitting algorithms
- Measurements done on [NUC]

Clad. Correctness

Cancellation at $\frac{\partial F}{\partial \gamma}$ for value of 2.

```
inline double breitwigner_pdf(double x, double gamma, double x0 = 0) {
  double gammahalf = gamma/2.0;
  return gammahalf/(M_PI * ((x-x0)*(x-x0) + gammahalf*gammahalf));
}
```

Clad

```
auto h = new TF1("f1", "breitwigner");
double p[] = {3, 1, 2};
h->SetParameters(p);
double x[] = {0};
TFormula::GradientStorage clad_res(3);
TFormula* formula = h->GetFormula();
formula->GradientPar(x, clad_res);
printf("Res=%g\n", clad_res[2]);
```

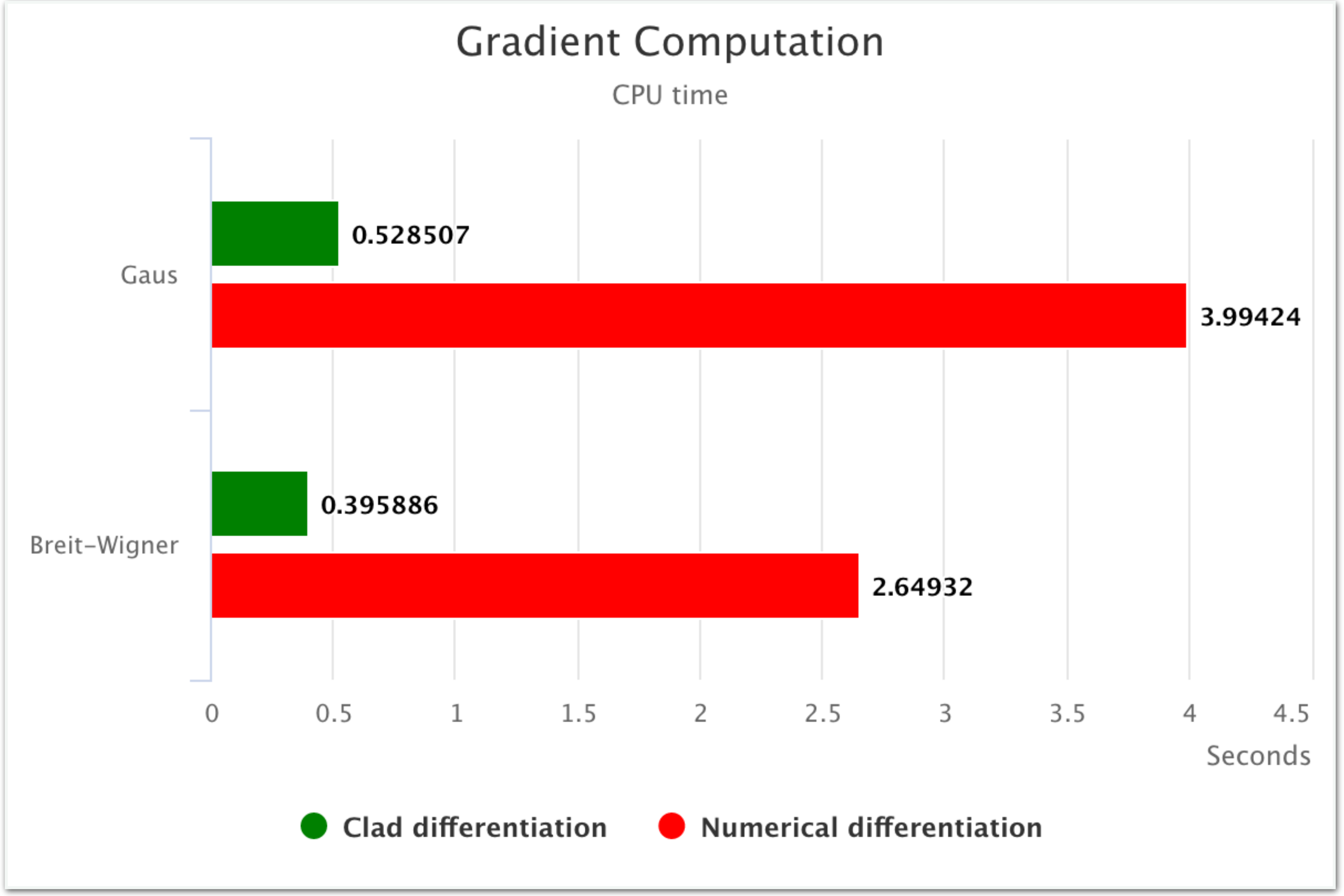
```
Res=0
```

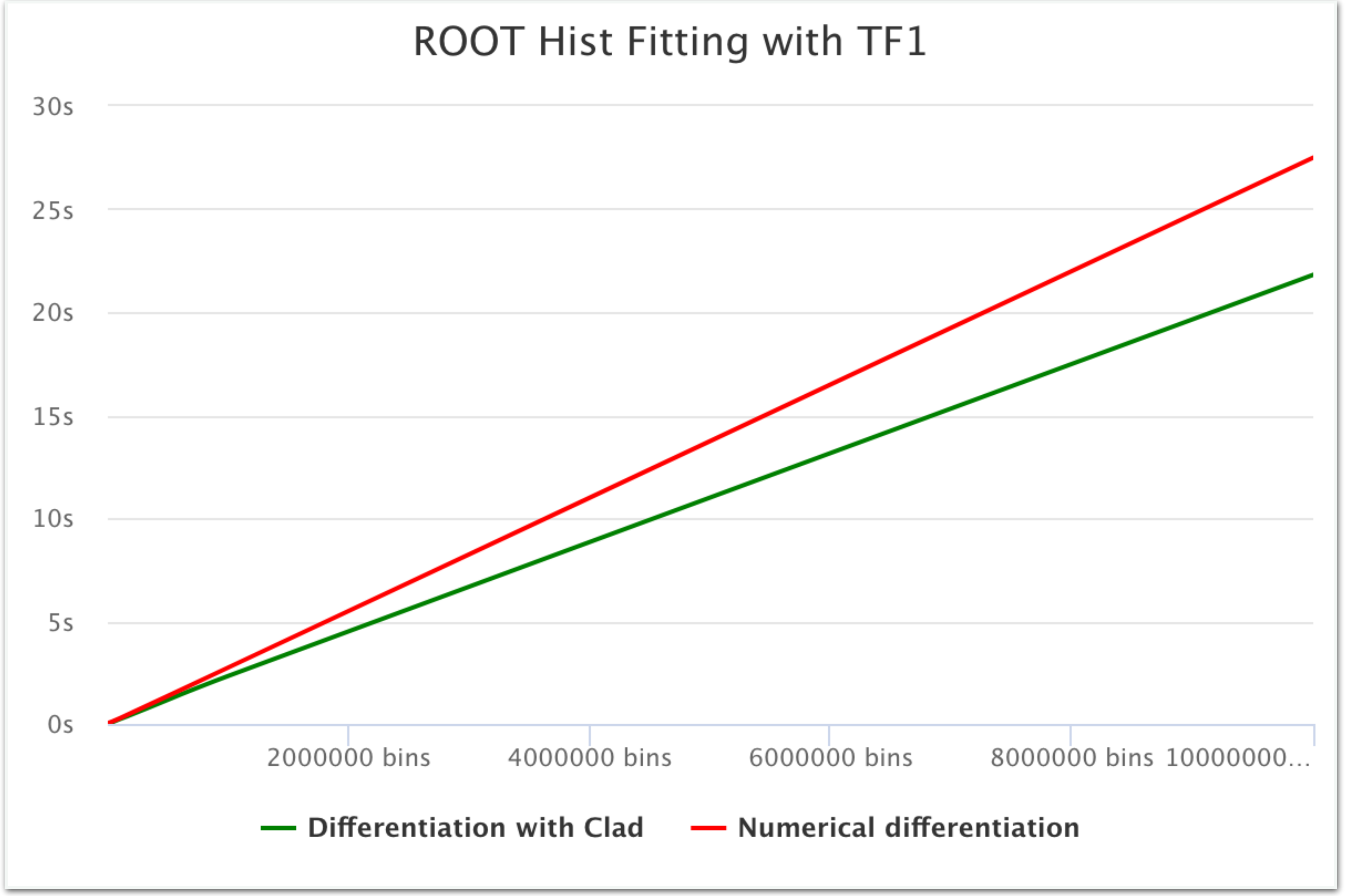
Numerical

```
auto h = new TF1("f1", "breitwigner");
double p[] = {3, 1, 2};
h->SetParameters(p);
double x[] = {0};
TFormula::GradientStorage numerical_res(3);
h->GradientPar(x, numerical_res.data());
printf("Res=%g\n", numerical_res[2]);
```

```
Res=-2.12793e-14
```

Clad. Results





The computation of gradient (on the left) shows significant benefits. We are investigating if we can project it in the ROOT fitting package (on the right) even better.

Clad. Results

Clad removes the iterations done by the numerical differentiation in **DoEval()**

Elapsed Time: 11.137s

- CPU Time: 7.620s
- Total Thread Count: 33
- Paused Time: 0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	Module	CPU Time
__GI___exp	libm.so.6	libm.so.6	3.534s
[Outside any known module]			2.753s
TFormula::GradientPar	libHist.so	libHist.so	0.228s
__dlopen	libdl.so.2	libdl.so.2	0.224s
memcmp	libc-dynamic.so	libc-dynamic.so	0.088s
[Others]			0.794s

Hotspots Insights

If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee view to track critical paths for these hotspots.

Explore Additional Insights

Parallelism: 25.0% (1.000 out of 4 logical CPUs)
Use Threading to explore more opportunities to increase parallelism in your application.

Microarchitecture Usage: 45.7%
Use Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.

Vector Register Utilization: 25.0%
Use Intel Advisor to learn more on vectorization efficiency of your application.

Elapsed Time: 56.082s

- CPU Time: 52.750s
- Total Thread Count: 33
- Paused Time: 0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	Module	CPU Time
__GI___exp	libm.so.6	libm.so.6	23.254s
[Outside any known module]			9.103s
TFormula::DoEval	libHist.so	libHist.so	5.076s
TF1::GradientParTempl<double>	libHist.so	libHist.so	3.040s
TF1::EvalPar	libHist.so	libHist.so	2.872s
[Others]			9.405s

Hotspots Insights

If you see significant hotspots in the Top Hotspots list, switch to th Bottom-up view for in-depth analysis per function. Otherwise, use Caller/Callee view to track critical paths for these hotspots.

Explore Additional Insights

Parallelism: 24.9% (0.998 out of 4 logical CPUs)
Use Threading to explore more opportunities to increase pa in your application.

Microarchitecture Usage: 61.5%
Use Microarchitecture Exploration to explore how efficientl application runs on the used hardware.

Vector Register Utilization: 25.0%
Use Intel Advisor to learn more on vectorization efficiency of y application.

Clad. Publications & Outreach

- Automatic Differentiation in C/C++ Using Clang Plugin Infrastructure, Lightening Talk at LLVM Dev Meeting, 17-18 Oct 2018, San Jose, CA, USA
- Successful Google Summer of Code project on "Extend clad - The Automatic Differentiation"

Clad. Future Work

- Continue advancing the automatic differentiation implementation
- Extend the usage of the TFormula differentiation backend
- Teach rootcling how to use clad and store the derivatives in the dictionaries

Code Modernization in ROOT. Optimize ROOT's I/O and dictionary format employing C++ Modules

Enable C++ Modules as a reflection dictionary provider

Completed Q3 Deliverable (available in ROOT v6.16 as a technology preview)

C++ Modules. Goals

- Improve correctness of ROOT
- Avoid parsing header files at ROOT's runtime
- Optimize performance of ROOT for third-party code (most notably ALICE, ATLAS, CMS and LHCb)
- Measurements done on [Vassil], [Yuka], [Oksana], [OpenLab]

C++ Modules. Correctness

Regular ROOT cannot load all C++ entities due to limitations of the implementation

```
[yuka@yuka-arch root-release]$ root -l
root [0] gMinuit
IncrementalExecutor::executeFunction: symbol 'gMinuit'
 unresolved while linking [cling interface function]!
```

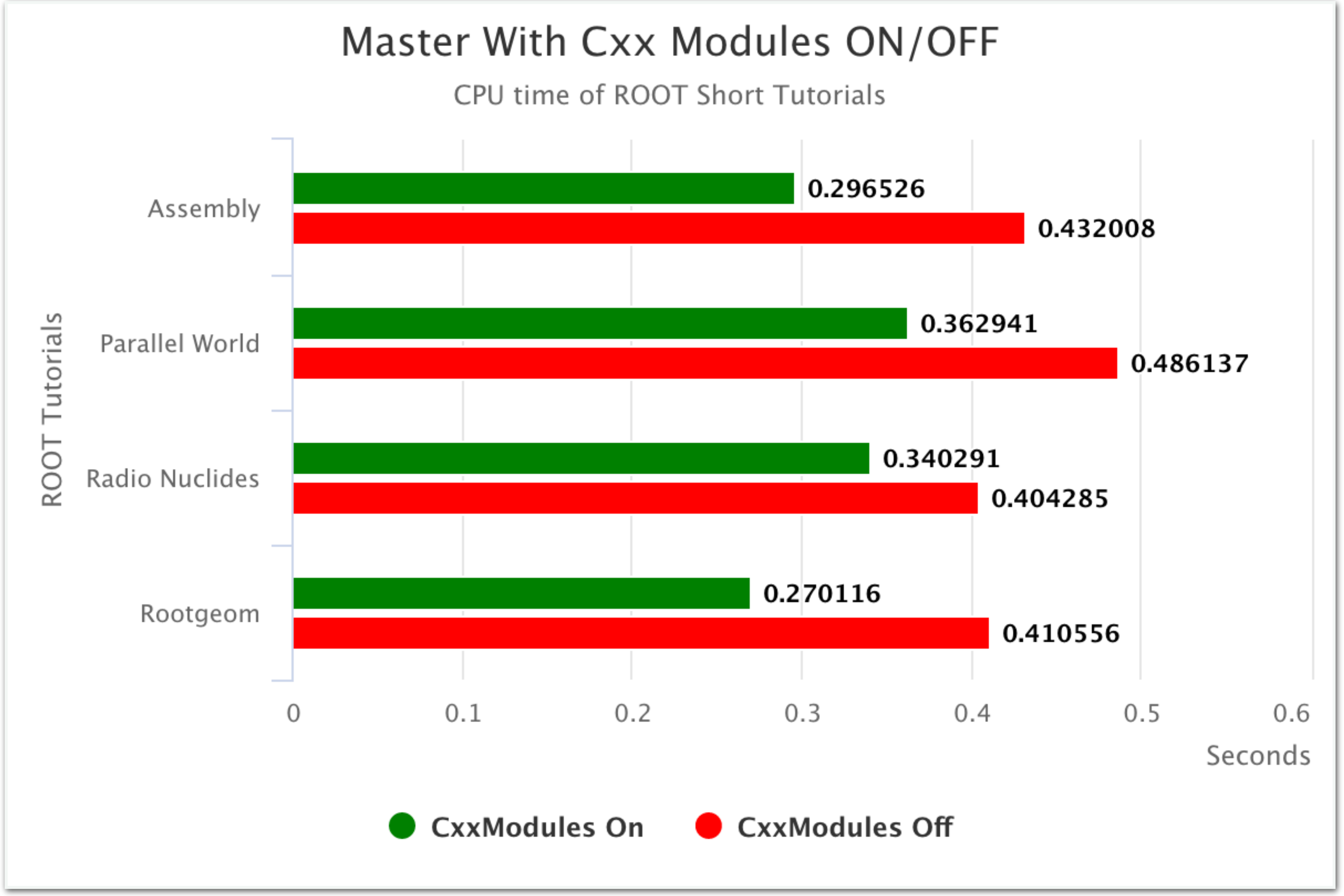
Using C++ Modules fixes the correctness issues.

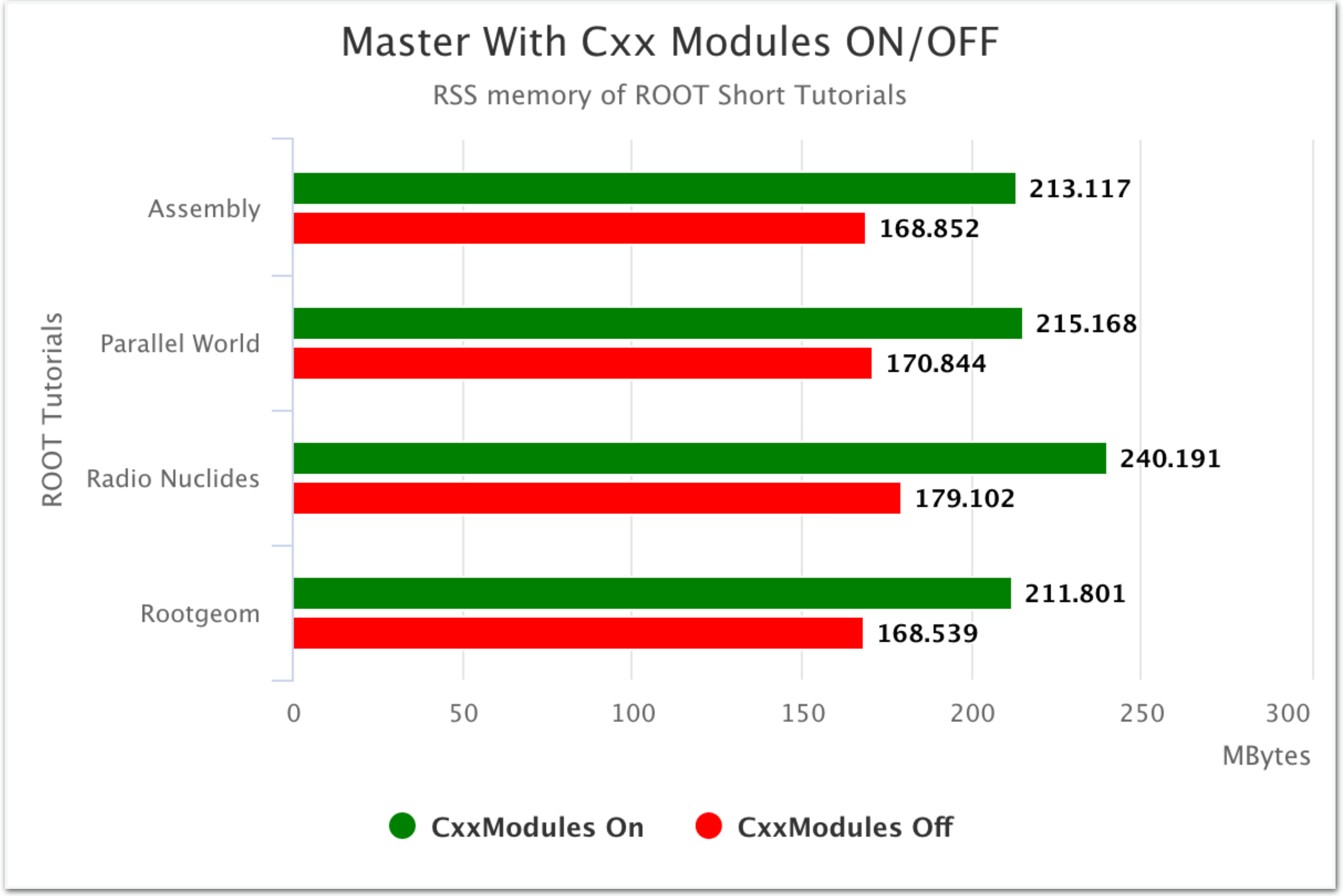
```
[yuka@yuka-arch module-release]$ root -l
root [0] gMinuit
(TMinuit *) nullptr
```

```
root [0] ROOT::Math::comp_ellint_1(0.2)
IncrementalExecutor::executeFunction: symbol '_ZN4ROOT4
Math13comp_ellint_1Ed' unresolved while linking [cling
interface function]!
You are probably missing the definition of ROOT::Math::
comp_ellint_1(double)
Maybe you need to load the corresponding shared library
?
root [1]
```

```
root [0]
root [1] ROOT::Math::comp_ellint_1(0.2)
(double) 1.5868678
root [2] .q
~/b/r/g/src (master ↩2=)
```

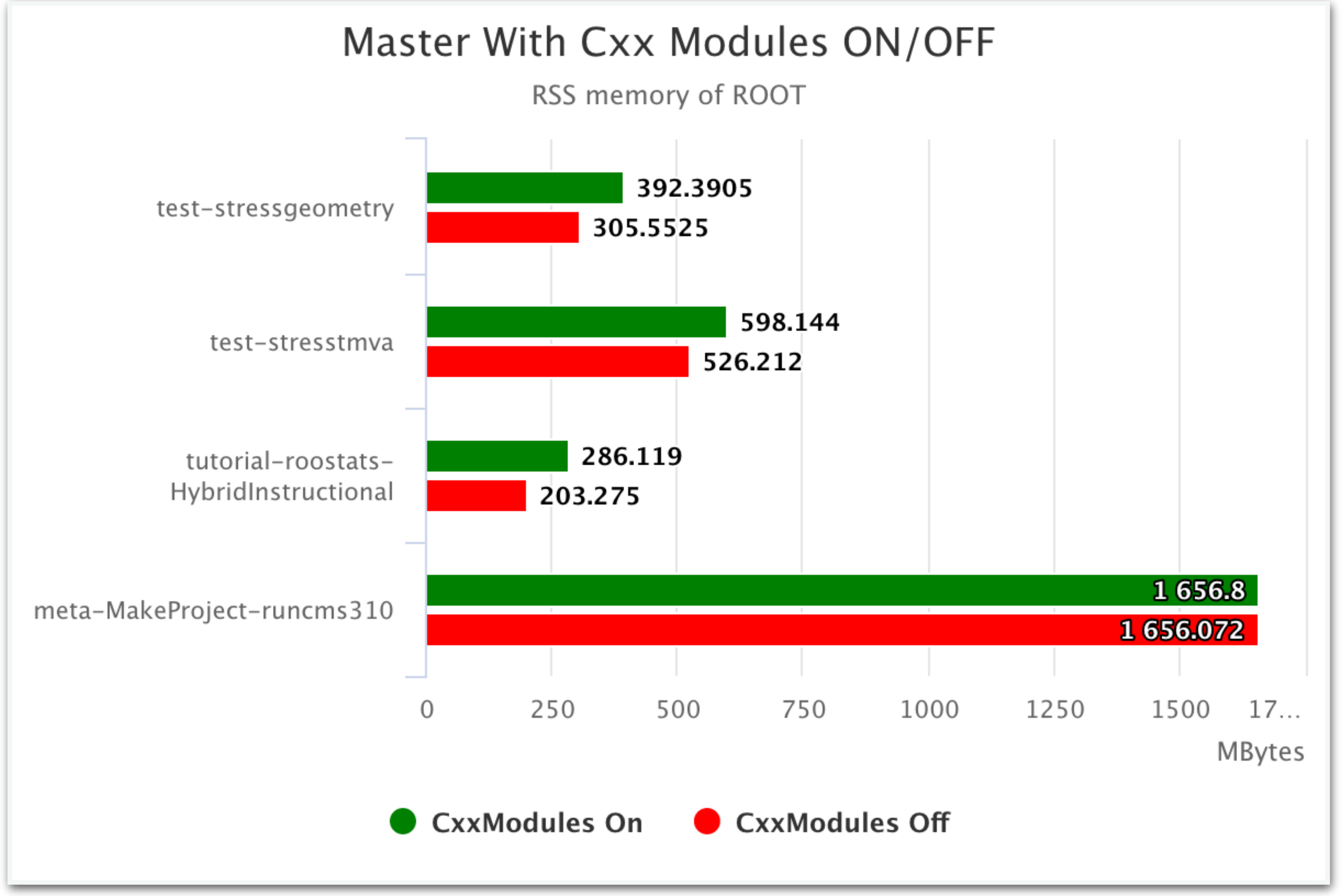
C++ Modules. Technology Preview



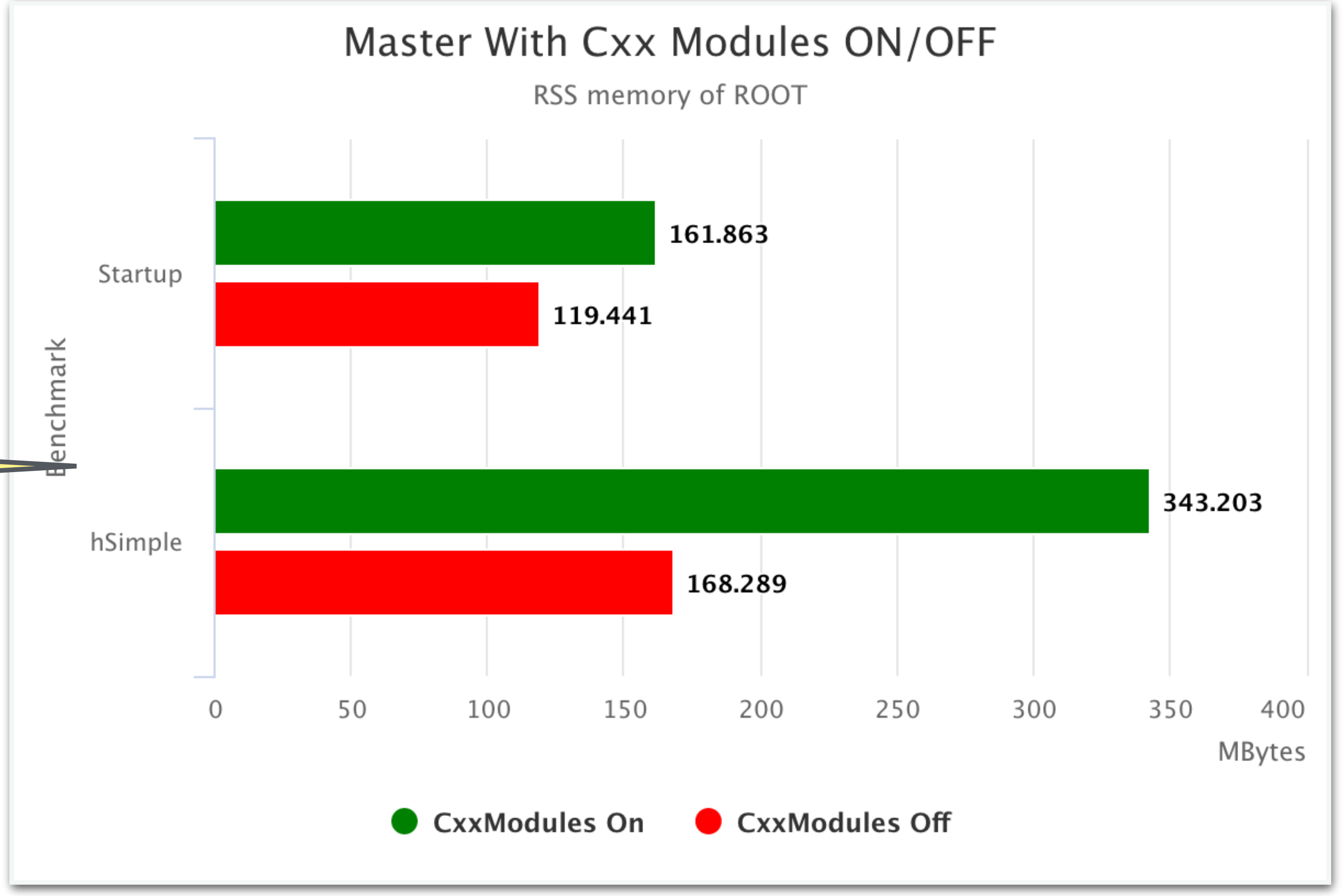


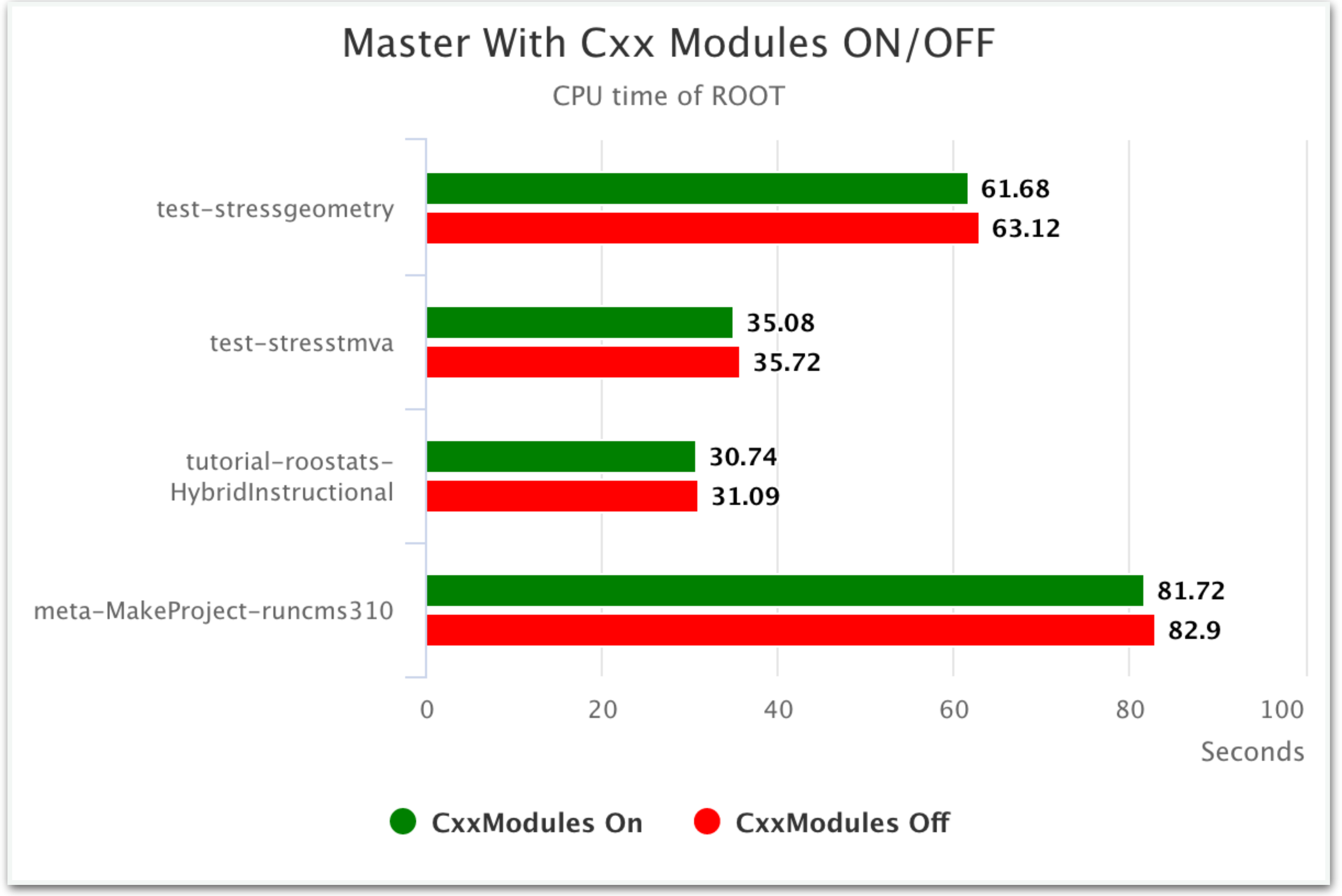
C++ Modules performance comparisons are made against ROOT's non-extendable optimization data structure (PCH). The major improvements will be in experiments' software stacks.

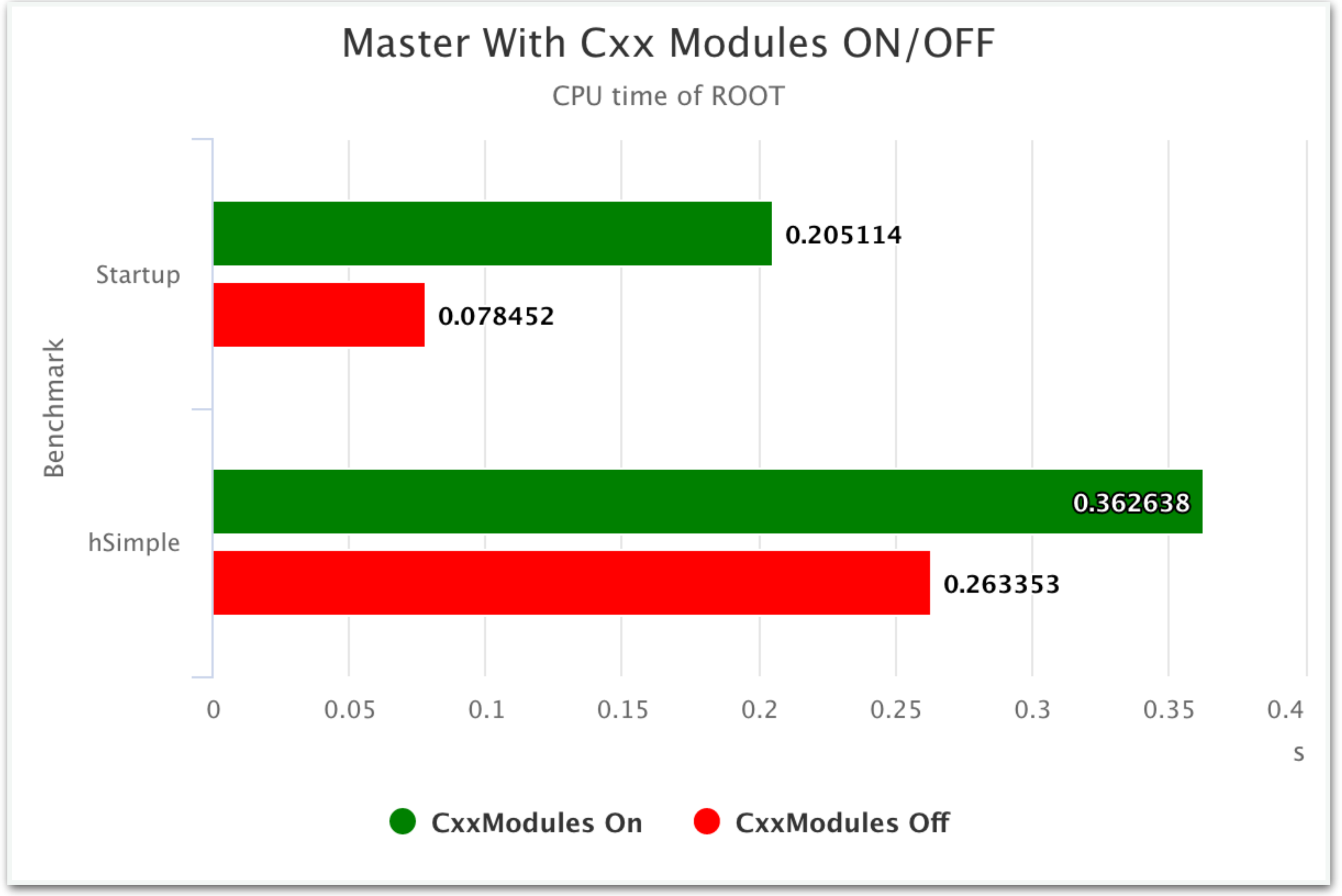
C++ Modules. Results



For small amount of work, we notice an overhead. It turns out to be a constant overhead introduced of the CxxModules preloading.

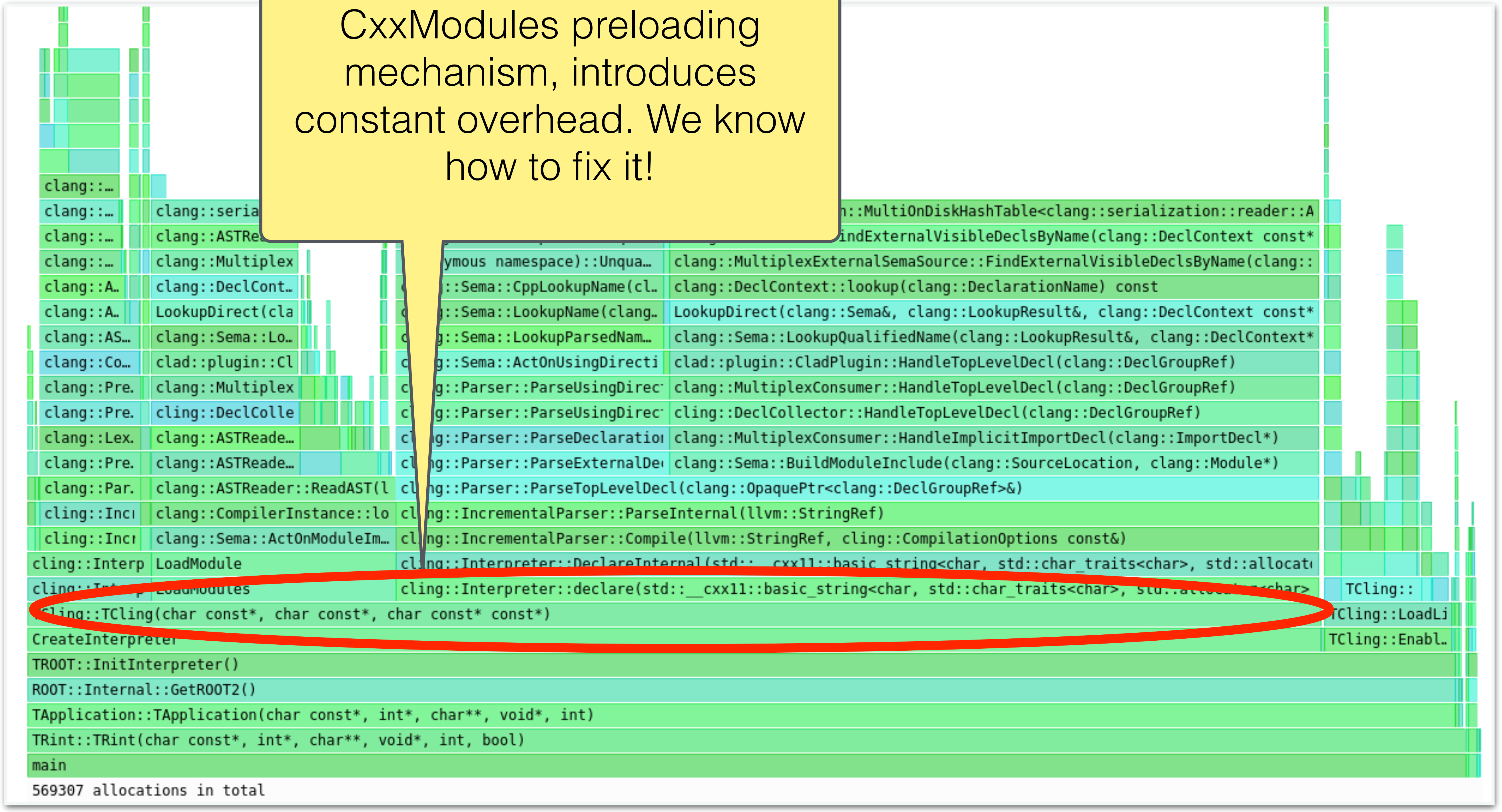


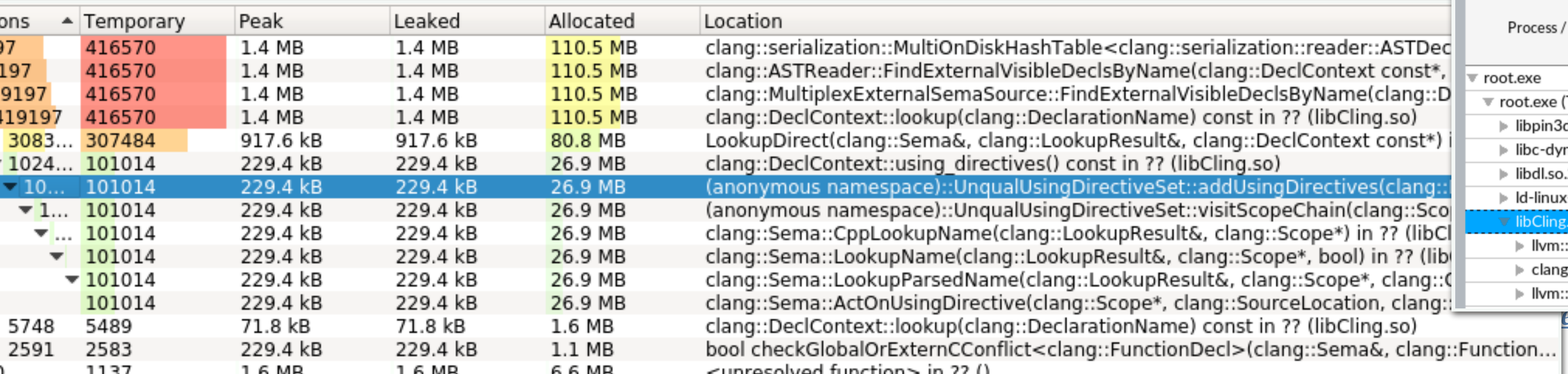




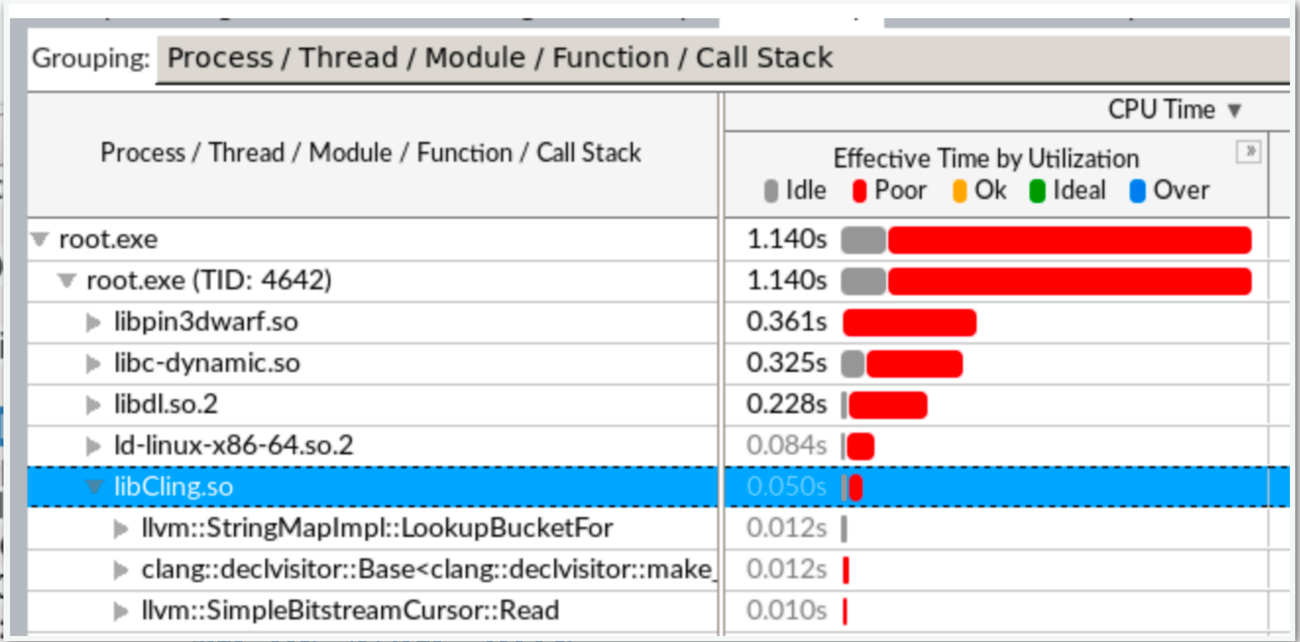
C++ Modules. Results

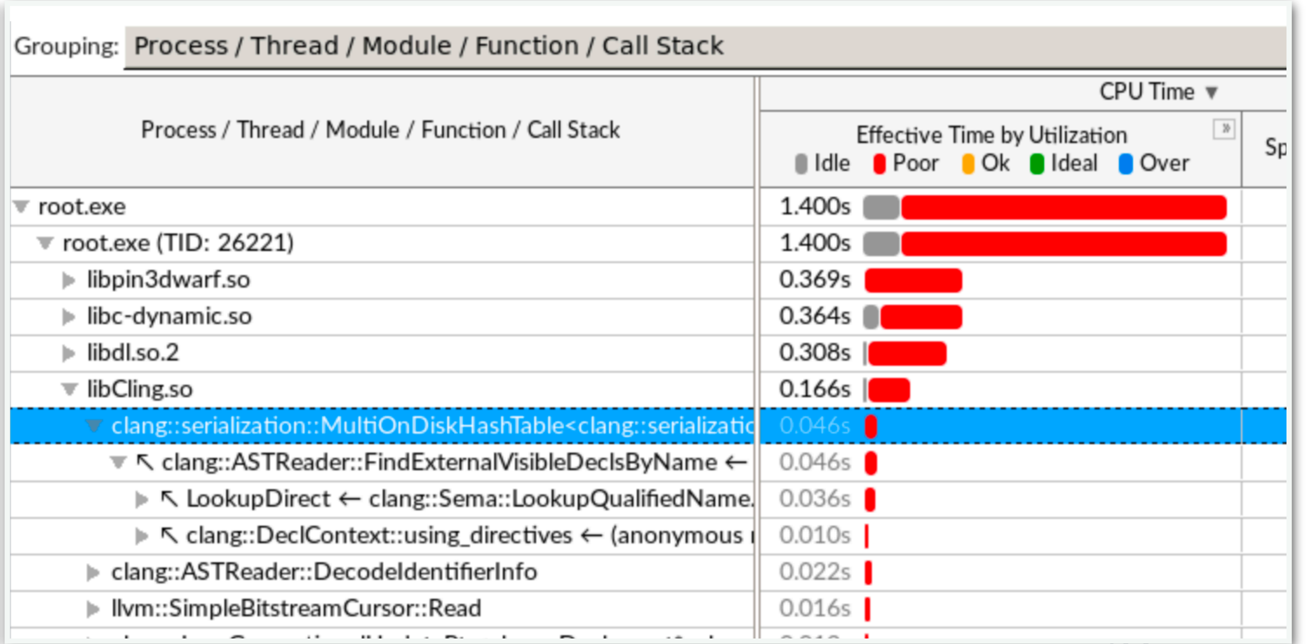
CxxModules preloading mechanism, introduces constant overhead. We know how to fix it!





ons	Temporary	Peak	Leaked	Allocated	Location
97	416570	1.4 MB	1.4 MB	110.5 MB	clang::serialization::MultiOnDiskHashTable<clang::serialization::reader::ASTDec
197	416570	1.4 MB	1.4 MB	110.5 MB	clang::ASTReader::FindExternalVisibleDeclsByName(clang::DeclContext const*,
9197	416570	1.4 MB	1.4 MB	110.5 MB	clang::MultiplexExternalSemaSource::FindExternalVisibleDeclsByName(clang::D
419197	416570	1.4 MB	1.4 MB	110.5 MB	clang::DeclContext::lookup(clang::DeclarationName) const in ?? (libCling.so)
3083...	307484	917.6 kB	917.6 kB	80.8 MB	LookupDirect(clang::Sema&, clang::LookupResult&, clang::DeclContext const*) i
1024...	101014	229.4 kB	229.4 kB	26.9 MB	clang::DeclContext::using_directives() const in ?? (libCling.so)
▼ 10...	101014	229.4 kB	229.4 kB	26.9 MB	(anonymous namespace)::UnqualUsingDirectiveSet::addUsingDirectives(clang::
▼ 1...	101014	229.4 kB	229.4 kB	26.9 MB	(anonymous namespace)::UnqualUsingDirectiveSet::visitScopeChain(clang::Sco
▼ ...	101014	229.4 kB	229.4 kB	26.9 MB	clang::Sema::CppLookupName(clang::LookupResult&, clang::Scope*) in ?? (libCl
▼	101014	229.4 kB	229.4 kB	26.9 MB	clang::Sema::LookupName(clang::LookupResult&, clang::Scope*, bool) in ?? (lib
▼	101014	229.4 kB	229.4 kB	26.9 MB	clang::Sema::LookupParsedName(clang::LookupResult&, clang::Scope*, clang::C
	101014	229.4 kB	229.4 kB	26.9 MB	clang::Sema::ActOnUsingDirective(clang::Scope*, clang::SourceLocation, clang::
5748	5489	71.8 kB	71.8 kB	1.6 MB	clang::DeclContext::lookup(clang::DeclarationName) const in ?? (libCling.so)
2591	2583	229.4 kB	229.4 kB	1.1 MB	bool checkGlobalOrExternCConflict<clang::FunctionDecl>(clang::Sema&, clang::Function...
	1137	1.6 MB	1.6 MB	6.6 MB	<unresolved function> in ?? ()





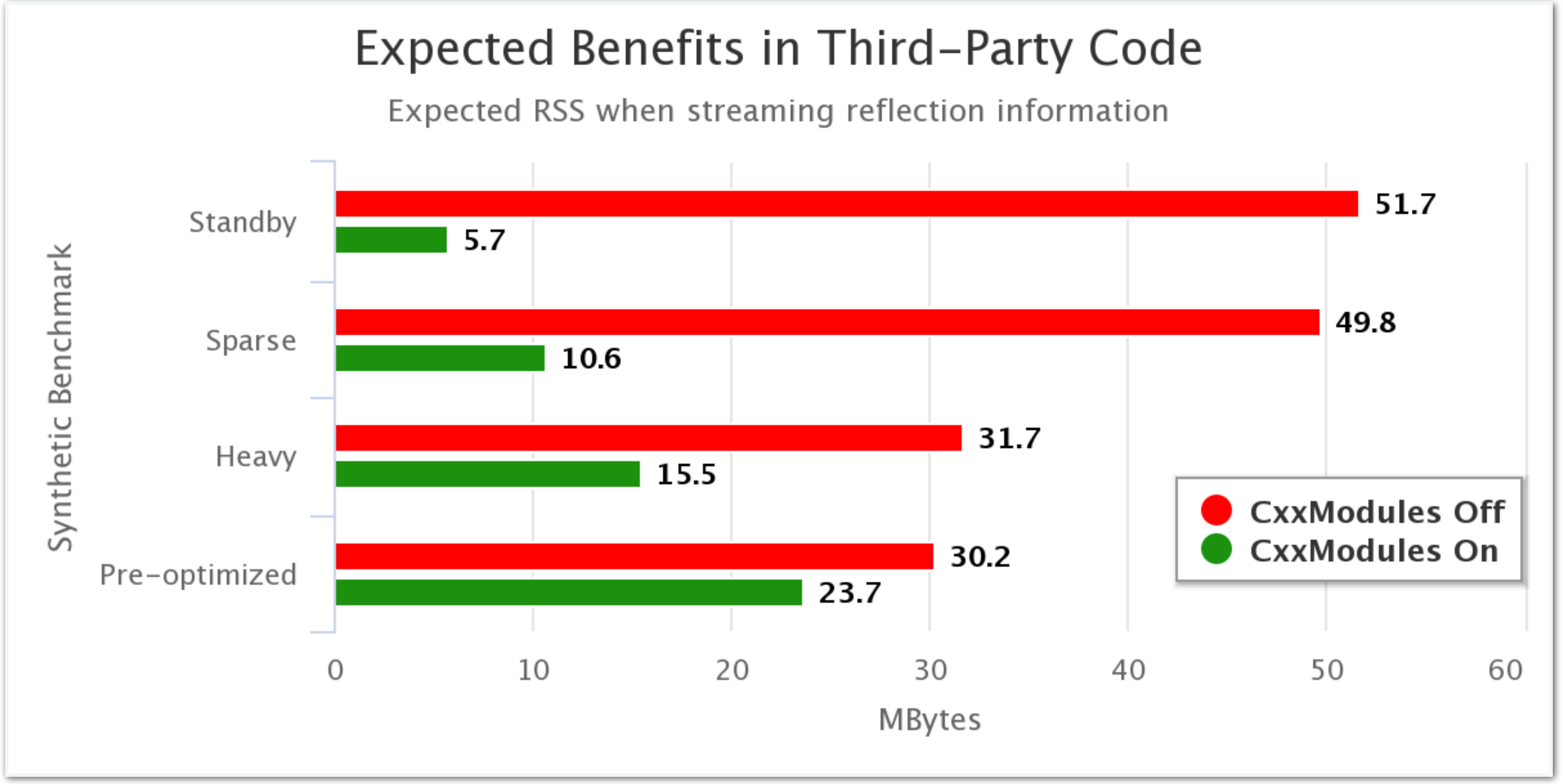
C++ Modules. Publications & Outreach

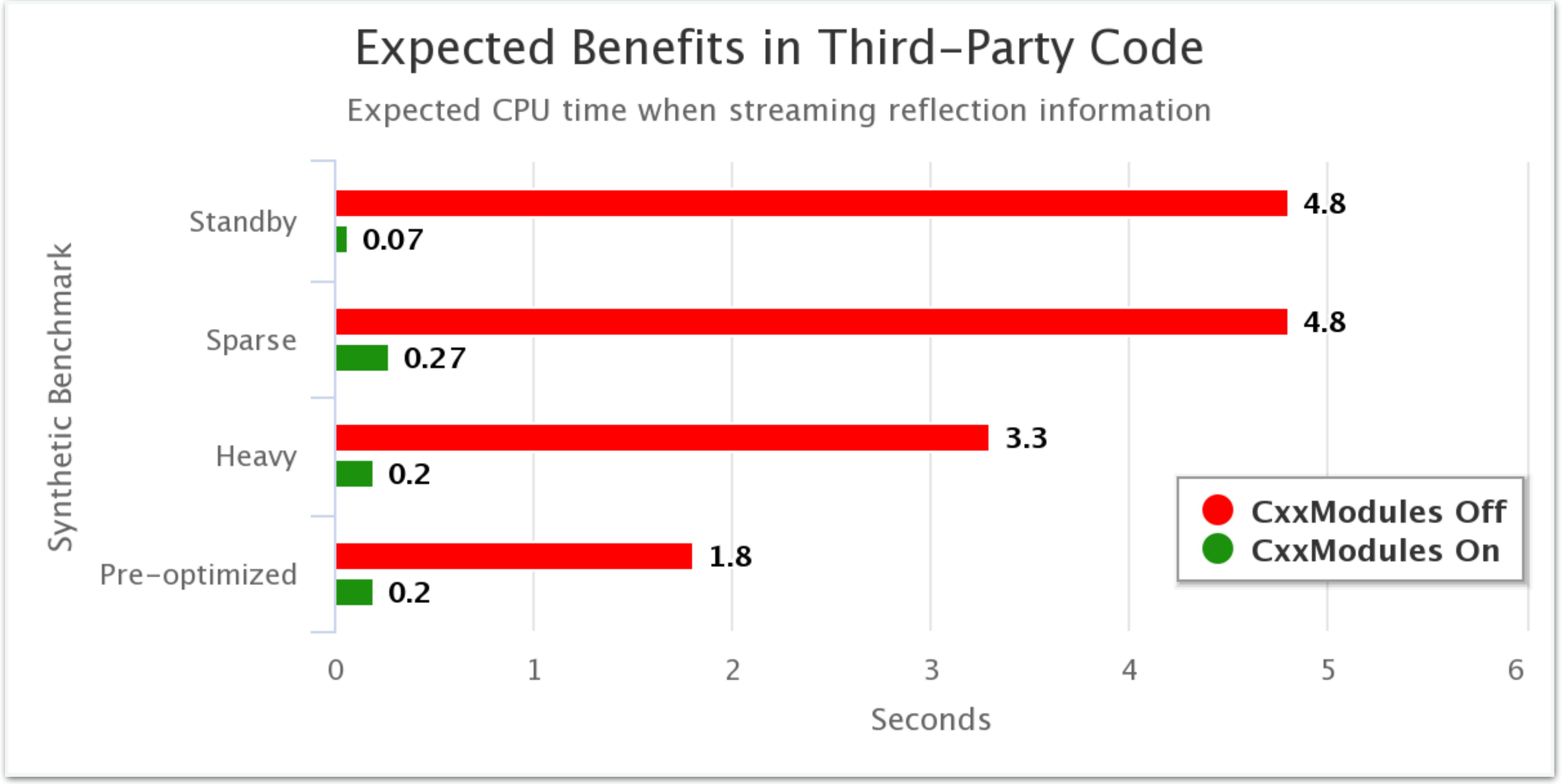
- Optimizing Frameworks' Performance Using C++ Modules-Aware ROOT , Poster at CHEP, 9-13 July 2018, Sofia, Bulgaria
- Collaboration with CMSSW for an early adoption of the feature (see GitHub meta issue)
- Various presentations in CERN-SFT group, ROOT team, DIANA-HEP and ROOT Workshop

C++ Modules. Future Work

- Turn on the feature by default for ROOT
- Optimize the feature towards various workflows
- Help with the migration process of the third-party code, and in particular the major LHC experiments (ALICE, ATLAS, CMS, LHCb)

C++ Modules. Future Work





Synthetic benchmarks (on information not available in the PCH of ROOT) show promising results. We need to reconfirm once we deploy the technology in the experiments' software.

Code Modernization in ROOT. Optimize ROOT's reflection layer

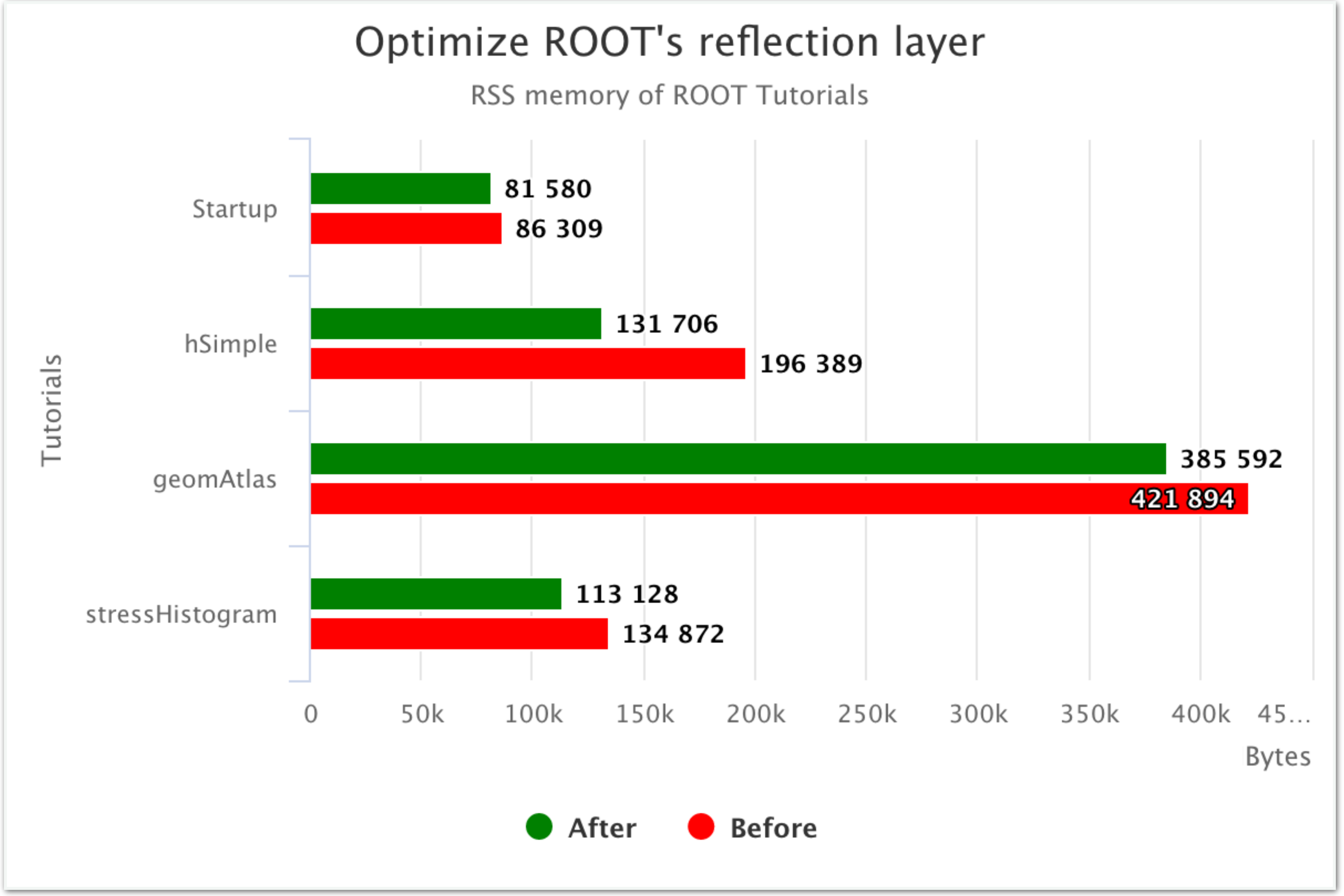
Reduce ROOT′s locking times

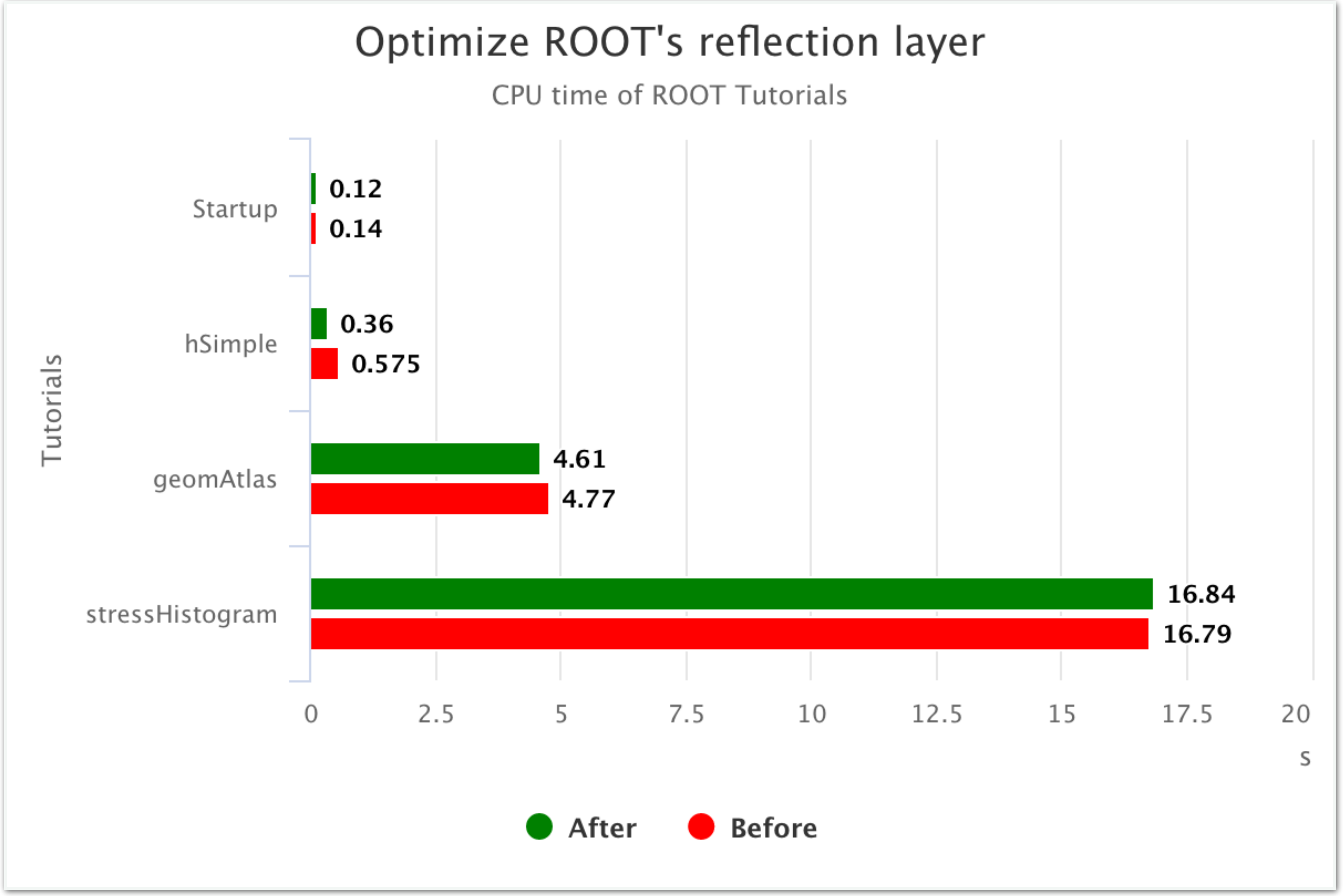
Completed Q4 Deliverable (available in ROOT v6.14 and ROOT v6.16)

Optimize ROOT's reflection layer. Goals

- Replace performance-inefficient legacy interfaces
- Optimize in-process memory footprint
- Measurements done on [Vassil]

Optimize ROOT's reflection layer. Results





Depending on the workflow we get up to ~33% memory reduction without execution regressions

Optimize ROOT's reflection layer. Future Work

- Pinpoint and optimize the next set of bottlenecks in ROOT's reflection layer

Extra work items

Completed (available in <u>ROOT master</u>)

Extra Things Delivered by IPCC-ROOT

- Move ROOT closer to LLVM upstream — reduced the technical debt in ROOT by moving it to the LLVM mainline
- Contributions to C++20 standard — participated in ISOCpp Standardization Meetings. Most notably 'constexpr virtual' as per P1064R0 accepted in the C++20 working draft.
- Upgrade to LLVM 5.0 — switch the internal fork to newer and more stable version of LLVM
- Number of contributions to the Clang Frontend — implemented a few optimizations and bug fixes with respect to C++ Modules
- Implement plugin support in cling — implemented a plugin-extension engine in cling where user plugins can specialize further the interpretative behavior of cling (such example is clad).
- Co-chaired the CHEP Conference in Sofia, Bulgaria

Future Directions

- Sustainability of the products of this work will be provided by the ROOT team, and some elements will be picked up by the recently NSF-funded IRIS-HEP Software Institute (http://iris-hep.org)

- We are looking forward to continue collaborating with Intel!

Conclusions

- During the 2 year project we explored the full software-hardware stack of the modern machines. We demonstrated performance improvements in threading, vectorization, compiler switches, compiler technologies and high-level algorithms
- We would like to express our deepest gratitude to Intel and the IPCC program for giving us such an opportunity!

Other Activities & Outreach

Continuous efforts

Training — CoDaS-HEP school

A school on tools, techniques and methods for Computational and Data Science for High Energy Physics.

- Second edition took place in Princeton University 23-27 July 2018
  - 60 participants
  - Topics included: performance tuning and optimization, vectorization, parallel programming (T. Mattson/Intel), and machine learning and big data tools.
- NSF has provided funding to continue this school for another 5 years

Collaborating project — DIANA/HEP

An NSF-funded project focused on developing tools for the HEP analysis tools ecosystem (of which ROOT is a core element). DIANA/HEP has three broad goals: improving performance, increasing interoperability of HEP tools with the broader scientific software ecosystem and providing tools for collaborative analysis.

For the IPCC, the focus on performance is the relevant part. The IPCC will collaborate with DIANA (and the ROOT team) on I/O and probably (eventually) RooFit modernization.

Team: Princeton, U.Nebraska-Lincoln, U.Cincinnati, NYU

Website: http://diana-hep.org

Related projects — Parallel Kalman Filter Tracking

Charged particle tracking reconstruction is the key pattern recognition algorithm requiring modernization for parallel architectures and the challenges of the HL-LHC. This is an NSF-funded project which is aiming to modernize these algorithms for use by CMS and others at the HL-LHC.

For the IPCC project, it provides a key testbed and use cases for testing vectorization (e.g. Matriplex, VecGeom)

Team: Princeton, UCSD, Cornell

Website: http://trackreco.github.io

Thank you!

I'd like to thank Raphael Isemann, Aleksandr Efremov and the ROOT team for the help;

Thanks to Claudio Bellini and Klaus-Dieter Oertel from Intel for providing useful insights throughout the project;

Special thanks to Luca Atzori and CERN OpenLab for providing the cutting edge Intel infrastructure and technical support.

Backup Slides

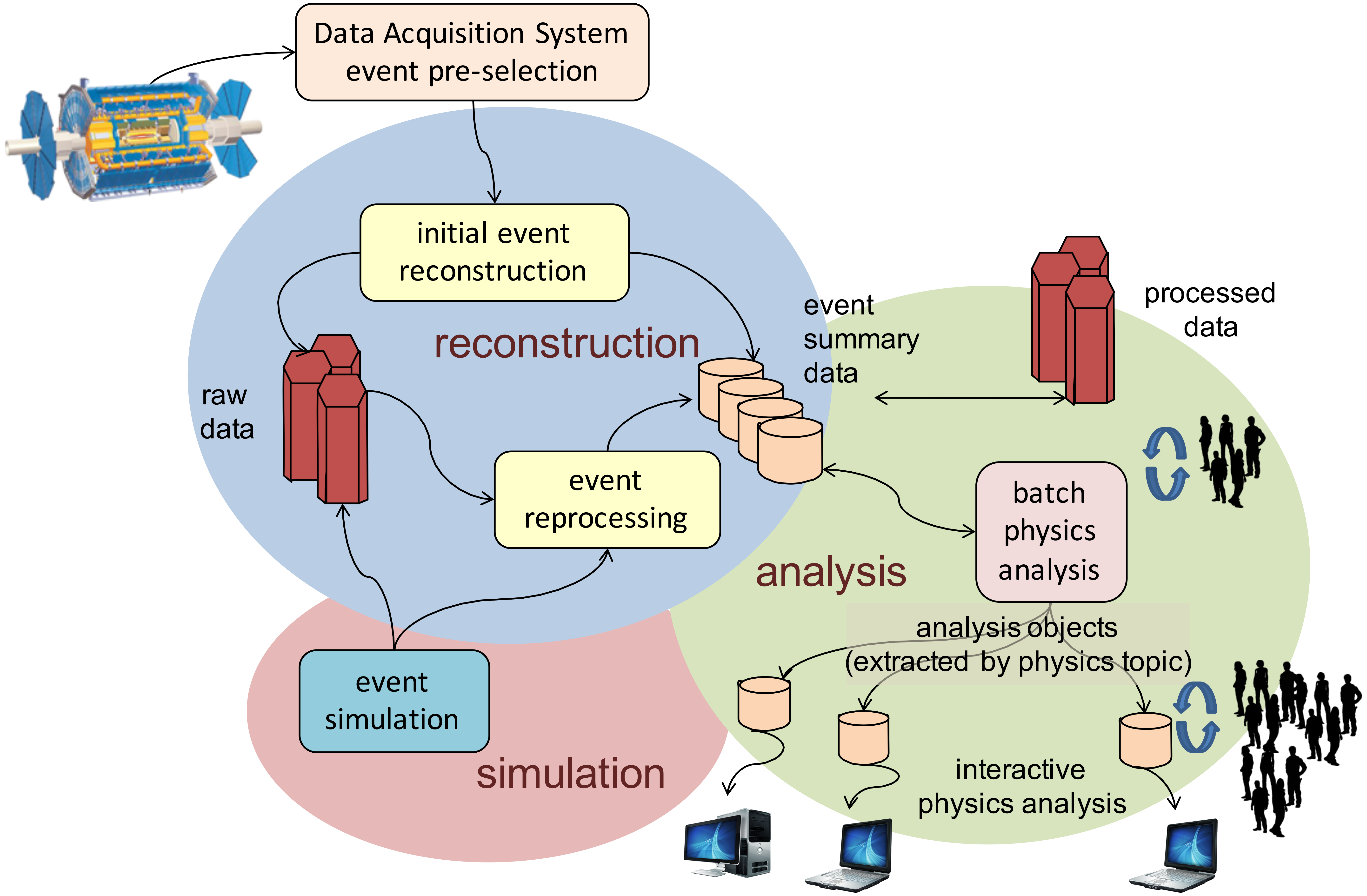
Might look messier than expected.

Further Reading About Clad

References:

[1] clad — Automatic Differentiation with Clang, http://llvm.org/devmtg/2013-11/slides/Vassilev-Poster.pdf
[2] clad Official GitHub Repository https://github.com/vgvassilev/clad
[3] clad demos https://github.com/vgvassilev/clad/tree/master/demos
[4] clad showcases https://github.com/vgvassilev/clad/tree/master/test
[5] More automatic differentiation tools http://www.autodiff.org/
[6] Automatic differentiation in Machine learning: a survey https://arxiv.org/pdf/1502.05767.pdf

Data Workflow



Worldwide LHC Computing Grid