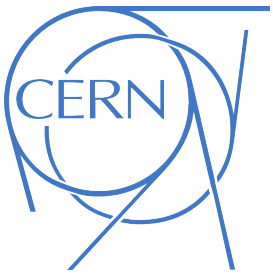

CERN Summer Student 2019

Project Report

Enhancements to ROOT performance benchmarking

Student:

Lilit Harutyunyan

Supervisors:

Oksana Shadura

Enrico Guiraud

July 26, 2019

Contents

1 Abstract

As software development grows in size and complexity, industry best practices include extensive quality testing often done at regular intervals or before a change is accepted into the codebase. ROOTBench is a continuous performance monitoring system for ROOT. This project sets out to enhance and improve ROOTBench by adding new feature (FlameGraphs) for inspecting the performance analysis of ROOT, extending new benchmarks and improving a library in rootbench.git.

2 ROOTBench

2.1 Introduction

Regular quality assurance processes for software development is widely referred to as continuous integration. Continuous integration often relies on fixed test suites and focuses on finding incorrect behavior.

As the field of high-energy physics (HEP) must deal with the large CPU and I/O costs of processing and analyzing billions of events, software performance is very important, in particular, for core libraries such as ROOT [1]. ROOTBench system provides:

- an extensible system tracking different performance metrics for the ROOT framework.
- speculative performance evaluation of proposed changes.
- rich and customizable visualizations and aids for performance analysis.

2.2 Technology

ROOTBench [2] is based on Googlebench micro benchmarking infrastructure [3]. Microbenchmarks focus on a single function or small piece of functionality instead of monolithic macro benchmarks. A micro benchmark provides a routine which checks performance only for some concise, targeted functionality. The call stack and code coverage of the benchmark should be as small as possible. It can be challenging for a micro-benchmarking infrastructure to provide stable measurements over time, especially as this may

require the same physical machines be consistently used for an extended time (to provide the broadest set of historical data).

ROOTBench uses Grafana for visualization because of its offered feature set and because the host lab for ROOTBench, CERN, provides an excellent on-demand service for Grafana.

Moreover, InfluxDB is the time series database for ROOTBench given its native support in Grafana. The setup allows to store a sequence of measured data points, providing data for the same performance measurements over time.

2.3 Information Flow

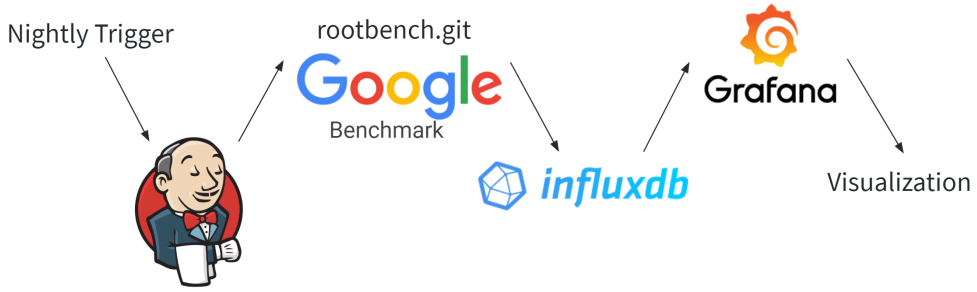

Figure 1: Information Flow in ROOTBench

The data measurements in the system starts from a set of user-defined benchmarks written using the Google benchmark framework and compiled to a standalone executables. All benchmarks are compiled by a nightly job using the Jenkins framework. The Jenkins job will compile both the ROOT framework and the ROOTbench software repository against the ROOT build.

After the builds have completed successfully, the set of benchmarks is in sequential mode on a dedicated machine (done to reduce the performance fluctuations). After each benchmark, the performance data measurements are uploaded into an InfluxDB database. The visualization system is only coupled to the continuous measurements via the database as shown in Figure 1. Grafana reads the database and visualizes the obtained data, making it available to the developer community via the Grafana web interfaces.

3 New Benchmarks in ROOTBench

In performance analysis of ROOT it is mostly interesting to see:

- Wall clock time
- User CPU time
- Memory measurements
- Stack traces

RDataFrame

RDataFrame offers a high level interface for analyses of data stored in TTrees, CSV's and other data formats [4]. As RDataFrame is the go-to ROOT analysis interface having benchmarks for it in ROOTBench would be very beneficial for analyzing its performance.

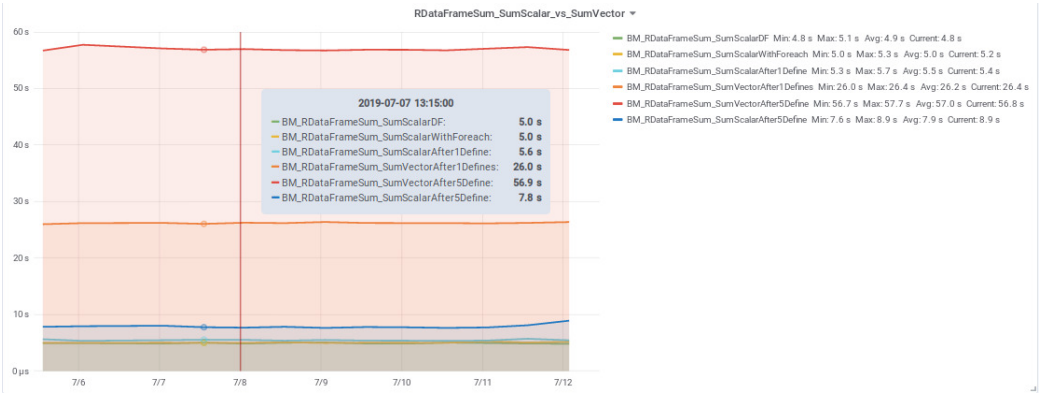

Figure 2: Plot from Grafana RDataFrame Dashboard

I extended 14 new RDataFrame and 12 TTreeReader and TBranch benchmarks in rootbench.git. RDataFrame benchmarks measure different functionality for the dataframe using the simpliest operation it offers - Sum(). For instance, *BM_RDataFrameSum_SumScalarWithForeach, BM_RDataFrameSum_SumScalarAfterXDefines* etc. (Figure 2).

I extended benchmarks also for TBranch and TTreeReader. They compare the performances between TTree::GetEntry(Long64_t event..) and

TTreePlayer::Next(). Some benchmarks about two different ways to do the summation operation both for RDataFrame and ROOT TTree/TreePlayer event loop readers were also added to rootbench.git.

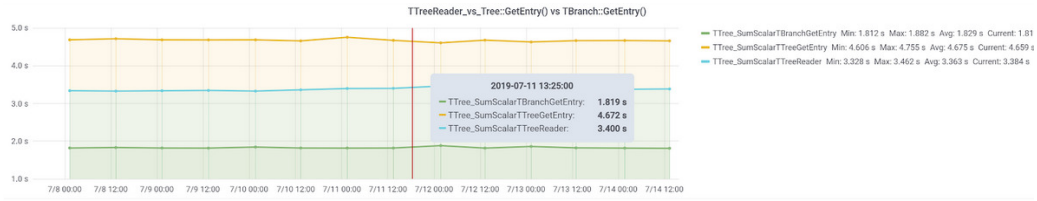

Figure 3: TTreeReader_vs_TTree::GetEntry() vs TBranch::GetEntry()

For example, from this visualization it is clear that the TTree_SumScalar TTreeGetEntry is the slowest. The reason for the overhead is the extra functionality that TTree::GetEntry() has (Figure 3). All the performance monitoring plots for the new benchmarks can be found in ROOTBench Grafana [5].

4 FlameGraphs

4.1 Introduction

FlameGraphs are interactive visualization of profiled software, allowing the most frequent code-paths to be identified quickly and accurately [6]. The benefits of FlameGraphs are that all data is in one picture, it is very easy to read and understand the stack traces and it is interactive when using JavaScript and browser.

There are different types of flamegraphs:

- CPU
- Memory
- Off-CPU
- Hot/Cold
- Differential

It is mostly interesting to see **CPU** and **memory** flamegraphs for ROOT.

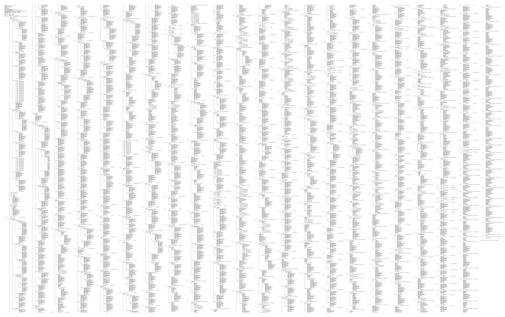

Figure 4: perf output

In a FlameGraph each box represents a function. Stack elements that are frequent can be seen, read and compared visually. Box width is proportional to the total time a function was profiled directly or its children were profiled.

4.2 Generation

1. Profile event of interest

The first step is to choose a profiler. One can use *perf, eBPF, SystemTap, Instruments, DTrace* etc. The profiled data is a series of stack traces. This profiled data is hard to read and can be thousands of lines long. It is often difficult to understand how stacks work when looking at this data (Figure 4).

2. stackcollapse.pl

This converts already profiled data into a single line of code. The full output consists of many lines, one line per stack. In addition, *grep* can be used to filter stacks before converting it into a FlameGraph.

3. flamegraph.pl

In the final step folded stacks are converted into interactive SVGs (Figure 5).

So we get full control of output, very high density details about the event, clear, easy to understand and interactive visualizations.

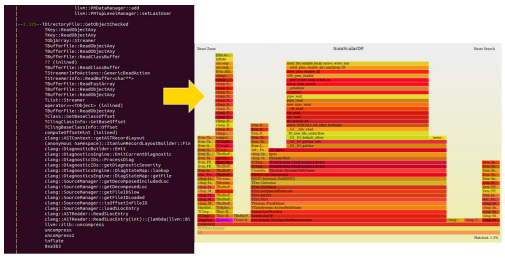

Figure 5: perf output to FlameGraph

4.3 CPU FlameGraphs

CPU FlameGraphs measure code paths that consume CPU. They are helpful in understanding and optimizing CPU usage, improving performance and scalability. It is commonly performed by sampling CPU stack traces at a timed interval.

Each box in a FlameGraph represents a *stack frame*. The **y-axis** shows stack depth. The top box shows the function that was on-CPU. Everything beneath that is ancestry. The function beneath a function is its parent.

The **x-axis** spans the sample population. It does not show the passing of time from left to right, as most graphs do. From left to right it is sorted alphabetically to maximize frame merging. Code paths split sometimes into different branches, which can reveal the key functions (Figure 6).

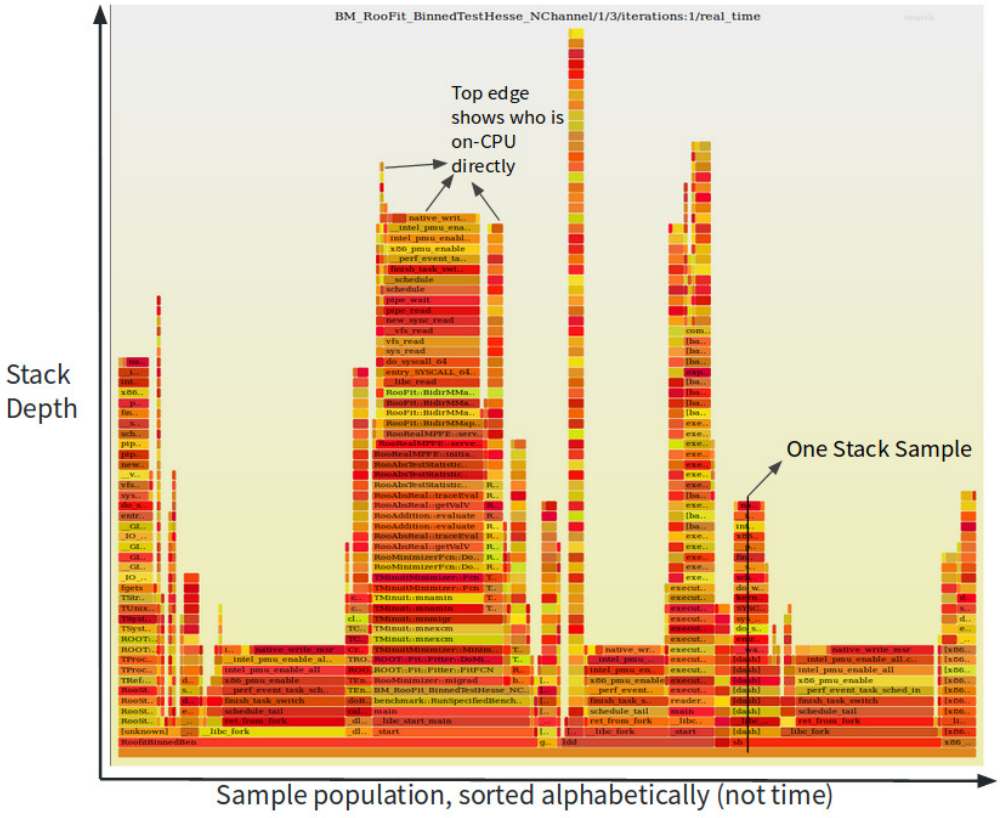

Figure 6: Information Flow in ROOTBench

FlameGraphs are very helpful when one wants to understand the extra overhead in different events. Here is an example of a FlameGraph for BM_TTreeSum_SumScalarTTreeReader, from which the overhead is obvious (Figure 7).

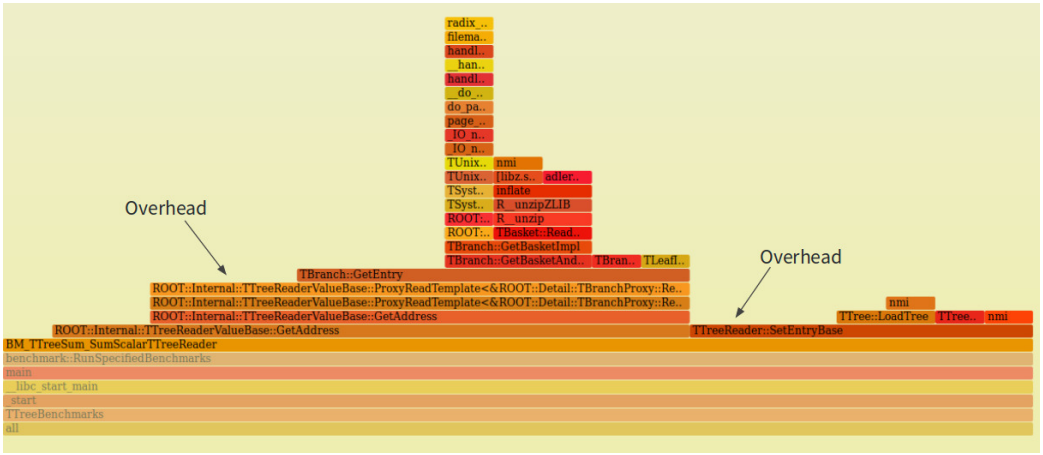

Figure 7: BM_TTreeSum_SumScalarTTreeReader FlameGraph

4.4 Memory FlameGraphs

Memory FlameGraphs are helpful when analyzing memory growth or leaks by tracing one of the following memory events

- Allocator functions: malloc(), free()
- brk() syscall
- mmap() syscall
- Page faults

Here instead of stacks and sample counts we measure stacks with byte counts. The steps of generation of FlameGraphs are the same, the only difference is the new option given to *perf record* command. I added the option to generate also memory FlameGraphs in rootbench.git (Figure 8).

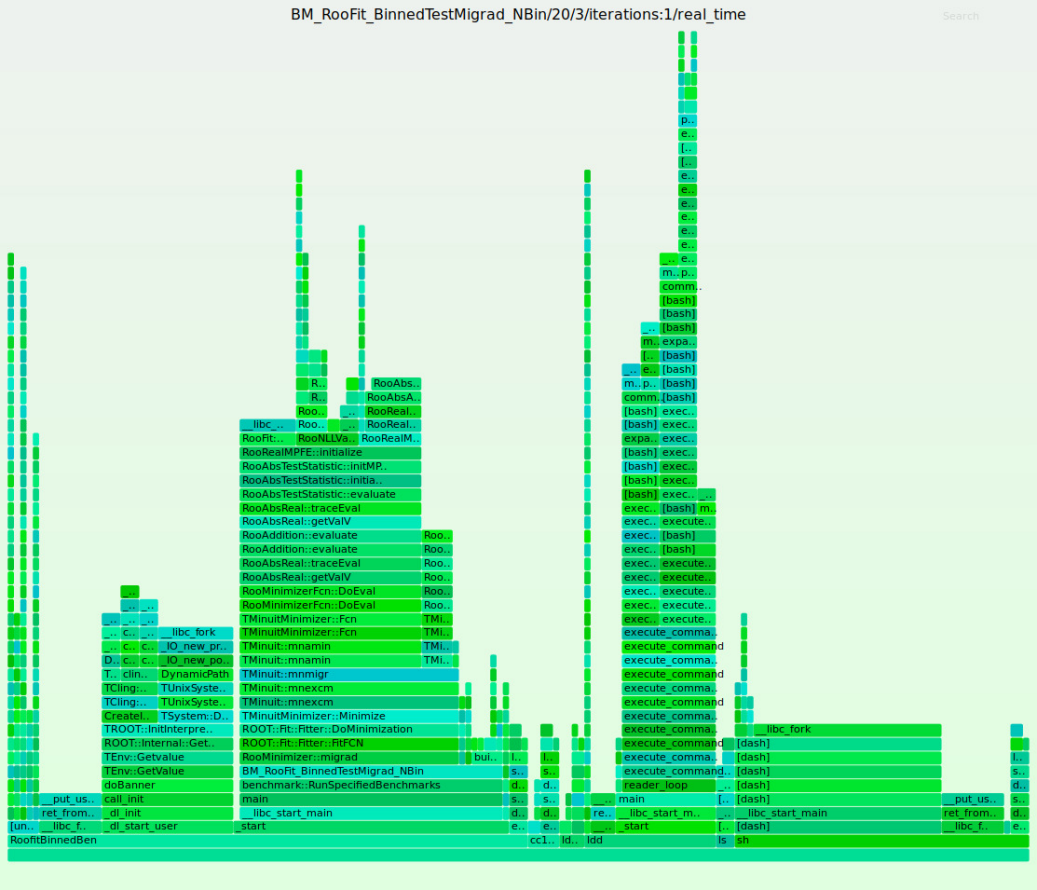

Figure 8: Memory FlameGraph for RooFit benchmark

4.5 Visualization

Now when we already have the new feature in ROOTBench, we want to display the generated FlameGraphs for easy access. There are two approaches:

- Generate them locally

  To generate the FlameGraphs locally one can use the *flamegraph.sh* script in rootbench.git. It has different options for generating the graphs. One can choose to generate them only for CPU, only for memory, or both. The FlameGraphs will be stored locally in the "build" directory of rootbench.git. [7]

- Grafana SVG panel

The FlameGraphs can be displayed also in Grafana SVG panel. This is convenient because ROOTBench uses Grafana to display other performance monitoring plots.

5 Improvements in RBSupport library

I have done some improvements in the RBSupport library. I added memory measurements for interpreted PyROOT and improved the memory measurements for ROOT interpreter. [8]

6 Future Improvements

This project can be extended as there are many different features that could be added to ROOTBench in order to have more enhanced performance analyses for ROOT. One of them could be to display the generated FlameGraphs in a page from where they can be viewed and downloaded. Another improvement would be to enable the current infrastructure to work on every ROOT pull request.

7 Conclusion

In conclusion, the project was very interesting and beneficial for me. ROOTBench is an important system for ROOT, which can be further developed and improved. I learned a lot during those eight weeks at CERN by working on my project, attending the lectures and visiting different sites of CERN. I got more motivated to continue my studies in computer science. I learned a lot from ROOT team members and my supervisors who helped me with the problems I faced. These two months at CERN were unforgettable and life-changing. I am very grateful for this opportunity.

References

[1] GitHub. 2019. GitHub - root-project/root : The official repository for ROOT: analyzing, storing and visualizing big data, scientifically. Available at: https://github.com/root-project/root. [Accessed 25 July 2019].

[2] GitHub. 2019. GitHub - root-project/rootbench: Collection of benchmarks and performance monitoring applications. Available at: https://github.com/root-project/rootbench. [Accessed 25 July 2019].

[3] GitHub. 2019. GitHub - google/benchmark: A microbenchmark support library. Available at: https://github.com/google/benchmark/. [Accessed 25 July 2019].

[4] RDataFrame. 2019. ROOT::RDataFrame Class Reference. Available at: https://root.cern/doc/master/classROOT_1_1RDataFrame.html. [Accessed 25 July 2019].

[5] ROOTBench Grafana. 2019. Available at: https://rootbnch-grafana-test.cern.ch/. [Accessed 25 July 2019].

[6] GitHub. 2019. GitHub - brendangregg/FlameGraph: Stack trace visualizer. Available at: https://github.com/brendangregg/FlameGraph. [Accessed 25 July 2019].

[7] GitHub. 2019. GitHub - root-project/rootbench: Flamegraph generation for rootbench.git. Available at: https://github.com/root-project/rootbench/pull/94. [Accessed 25 July 2019].

[8] GitHub. 2019. GitHub - root-project/rootbench: Add memory measurements to interpreted PyROOT. Available at: https://github.com/root-project/rootbench/pull/95. [Accessed 25 July 2019].