PERFORMANCE ANALYSIS AND OPTIMIZATION OF MPDROOT

J. Buša Jr.[1,3], S. Hnatič[1,a], O.V. Rogachevsky[2]

[1] Mescheryakov Laboratory of Information Technologies, Joint Institute for Nuclear Research,20 Joliot-Curie, Dubna, Moscow Region, 141980, Russia

[2] Veksler and Baldin Laboratory of High Energy Physics, Joint Institute for Nuclear Research, 4 Baldin St., Dubna., Moscow Region, 141980, Russia

[3] Institute of Experimental Physics, Slovak Academy of Sciences, Watsonova 47, Košice, 04001, Slovakia

E-mail: [a] hnatics@jinr.ru

MPDRoot is the software framework for simulation, reconstruction and physics analysis of the simulated and experimental data for MPD experiment at NICA. It is planned to obtain ~ 10^8 events of heavy ion collisions for physics analysis, hence it is crucial to have the effective and efficient methodology of the systematic performance improvement of MPDRoot's backend. In this work, we present the analysis of timing and instruction performance of MPDRoot's reconstruction by benchmarks and the Callgrind profiler. We evaluate the feasibility of speeding up reconstruction by the reduction of method-call overhead and the possible benefit of optimizing the math library. Based on the obtained results we draw conclusions about necessary steps to be taken in the near future of MPDRoot's development.

Keywords: MPD, MPDRoot, benchmarking, code review, code quality, optimization

Ján Buša Jr, Slavomír Hnatič, Oleg Rogachevsky

1. Introduction

The architecture of MPDRoot is shown in Fig.1 and is composed of the three main parts [1]:

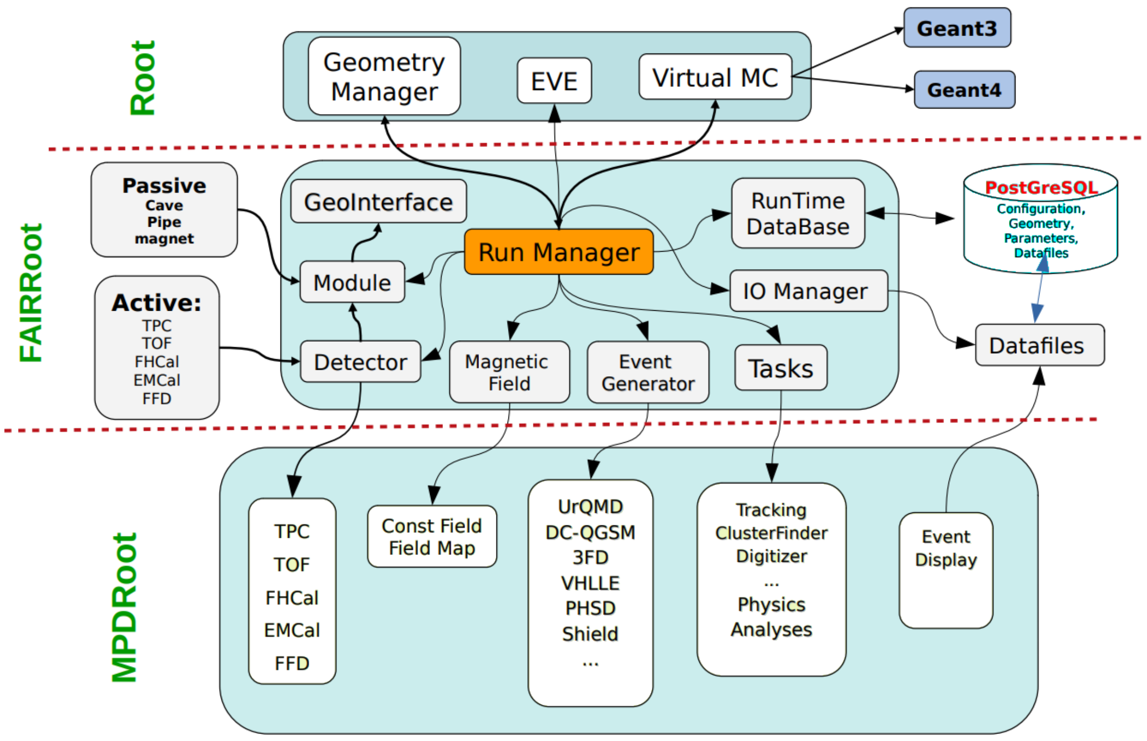


Figure 1. Architecture of MPDRoot

- Root – a set of building blocks and primitives written in C++ language, tailored for physics experiments
- FairRoot – simulation, reconstruction and analysis framework built on top of Root and the other set of packages encapsulated in FairSoft
- MPDRoot – specific implementation for the MPD experiment at NICA

2. Instruction and run-time profiling of MPDRoot

The process of optimization requires finding runtime bottlenecks, which is commonly achieved by measuring performance of various software entities in units of time and instructions. For this purpose, MPDRoot and FairSoft/FairRoot suite must be built with debug symbols. The information about the instruction profile of various parts of the MPDRoot's reconstruction is obtained by running the Callgrind tool from the Valgrind suite [2]. The output from the Callgrind profiler can then be visualized in a KCachegrind tool.

However, from the practical point of view, it is the physical time the software spends in its given entity, which represents the true performance measure. The accurate measurement of such performance is somewhat problematic as it uses expensive system clock calls. If often used, these skew results and the direct timings of low-cost software entities are usually completely invalid. The pie chart in Fig. 2 shows the difference between the instruction profiler results and the true physical time for the various tasks of the MPDRoot's reconstruction. This means, the instruction profile obtained from Callgrind is suitable for finding performance bottlenecks.

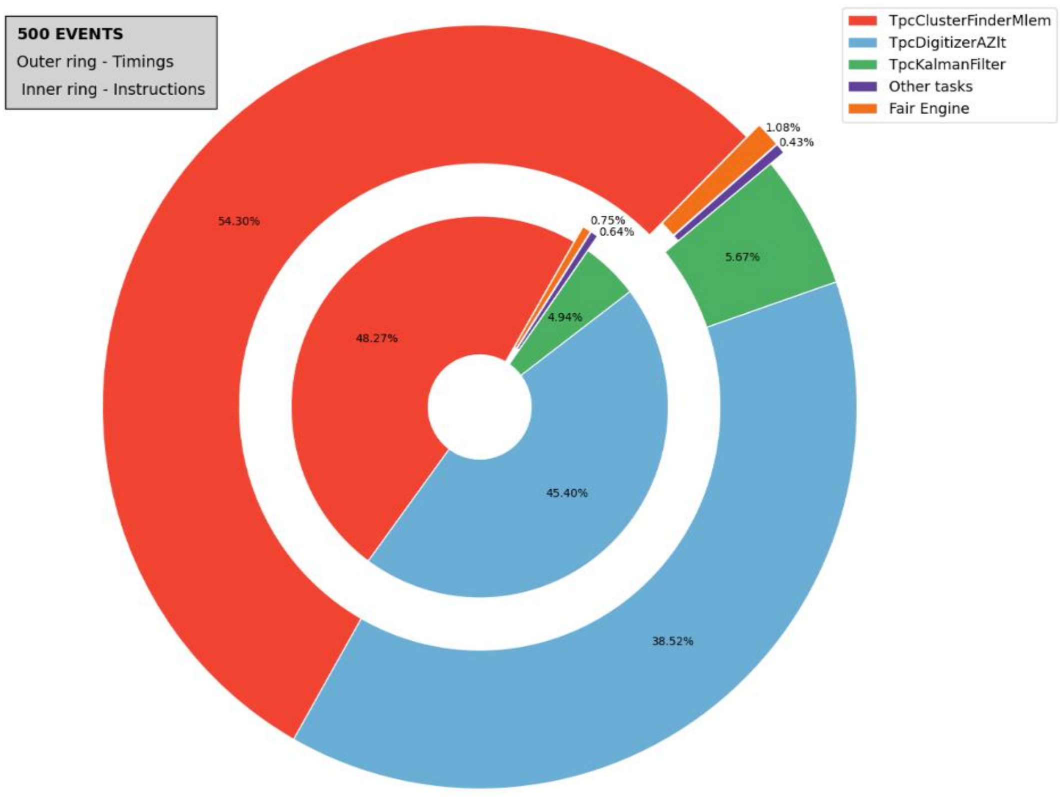


Figure 2. Time vs Instructions cost of MPDroot's reconstruction tasks

MPDRoot spends most of the time in digitizing (~45%) and clusterization (~48%) followed by the Kalman filter (~5%) and the Fair engine (less than 1%). Hence, it is logical to look into digitizer and clusterization algorithms for speedup optimization at the current stage of MPDRoot's development.

3. Reducing method-call overhead

It is possible to speedup the code by reducing the method-call overhead with the inline keyword (or the flatten attribute), however the result of code inlining is many times counterproductive [3]. In the most common case, the removal of the call stack results in a larger executable binary, which in turn slows down the execution time. The table in Fig.3 shows the effect of reducing the method-call overhead by inlining the most frequently called methods from Digitizer and ClusterFinder tasks.

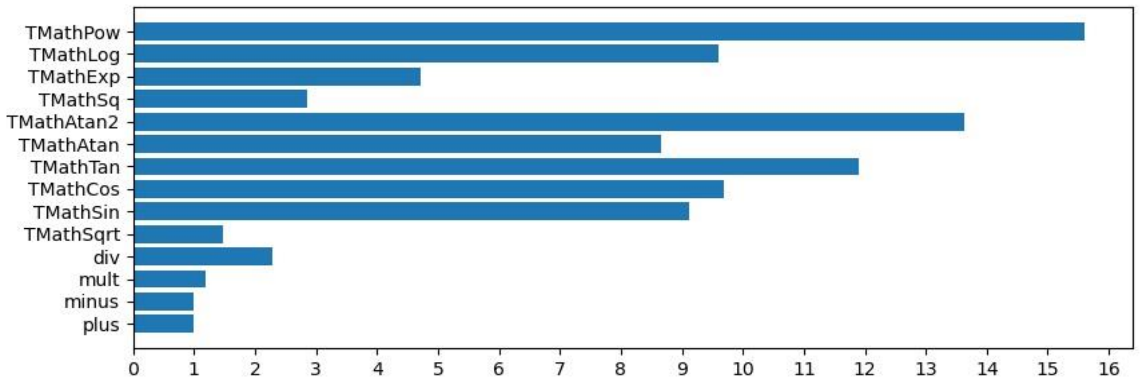
	% instructions out of total	instructions per call	% of calls inlined	task speedup	total speedup
CalcOrigin (Digitizer task)	4.8	18	100	4.2%	1.9%
GetCij (ClusterFinder task)	12	380	90	-1.2%	-6.3%

Figure 3. Effect of method inlining on MPDroot's performance

While inlining the cheap CalcOrigin method (18 instructions per call) results in an overall speedup by 1.9% of time, inlining the more expensive GetCij method (380 instructions per call) slows down the total runtime. Despite the obvious positive effect, inlining the CalcOrigin method at the current stage of development is not a good idea, as this method can be modified in the future.

4. Benchmarking and optimization of math methods

Such a closed for modification, well tested software entity used in MPDRoot is the TMath library. The results of the benchmark of TMath methods running in MPDRoot are presented in the upper plot of Fig.4. The power, logarithm, and trigonometric are the most expensive methods.



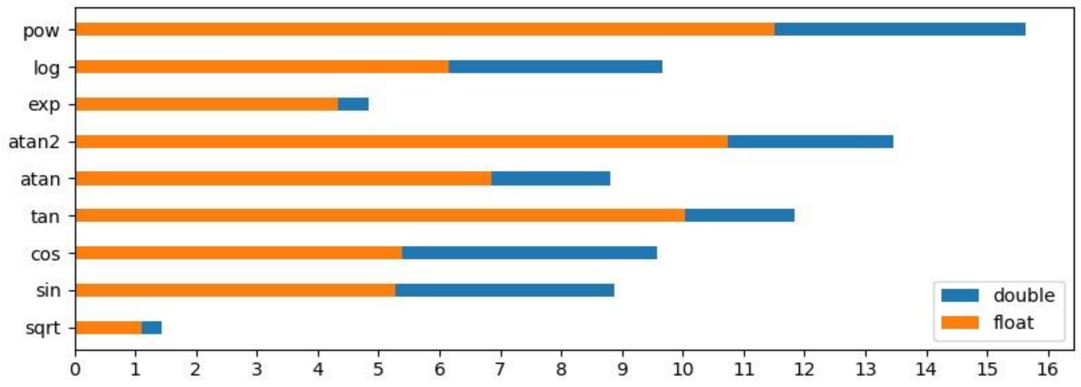


Figure 4. Math methods benchmarks in MPDroot.
Upper plot: cost of TMath methods in units of the '+' operation cost
Bottom plot: cost of double vs float precision methods

If one is careful with cumulative errors, the quickest way to achieve speedup of math methods is to replace default double precision methods with their float precision analogues. Thus, one can get up to 45% speedup (bottom plot of Fig.4).

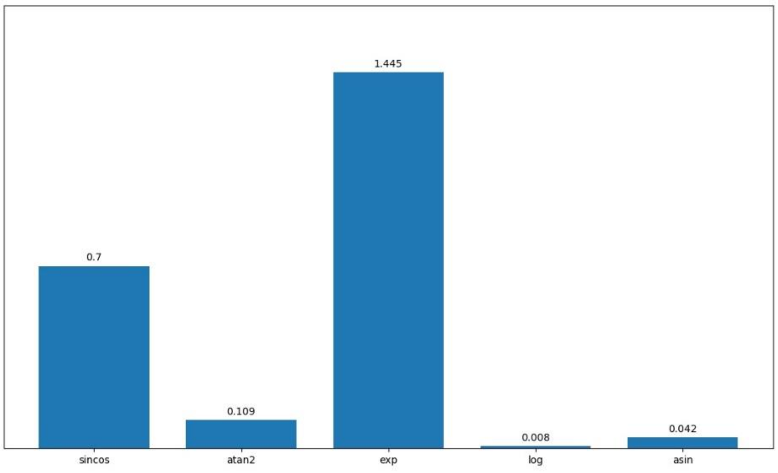


Figure 5. TMath methods instructions percentage in MPDroot

The plot in Fig.5 shows the instruction percentage of the five most present math methods in MPDRoot's reconstruction. Overall, the whole TMath library takes less than 2.5% out of the total number of instructions, therefore any optimization of math library is currently not justifiable as it will not lead to any significant reconstruction speedup.

5. Conclusions and near future perspectives

It is shown, that the effective methodology of MPDRoot's optimization consists of firstly isolating bottlenecks by timing benchmarks and instruction profiling. Out of those, the reasonable candidates for optimization are entities already closed for modification in the MPDRoot's development lifecycle. Having such software entities in the MPDRoot's codebase is necessary for effective optimization. This means it is crucial to focus on improving the software quality, which will in turn result in a well-tested modularized professional grade code [4], [5], [6].

We will implement the following changes to the MPDRoot's software development process:

1. Implementation of the code ownership feature - essential for the code review process.
2. Implementation of the QA tests engine – to minimize risks associated with algorithm logic changes or extensions
3. Implementation of the unit test engine – to minimize risks associated with system changes or extensions

6. Acknowledgements

The work was supported by the RFBR grant ("Megascience – NICA") No. 18-02-40102.

References

[1] Rogachevsky, O.V., Bychkov, A.V., Krylov, A.V. et al. Software Development and Computing for the MPD Experiment. Phys. Part. Nuclei 52, 817–820 (2021).

[2] https://valgrind.org/docs/manual/cl-manual.html (accessed 09.09.2021)

[3] https://isocpp.org/wiki/faq/inline-functions (accessed 09.09.2021)

[4] Robert C. Martin ("Uncle Bob"), Principles of OOD. Available at: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod (accessed 09.09.2021)

[5] K.J. Lieberherr, I.M. Holland, Assuring good style for object-oriented programs. IEEE Software. September 1989, pp. 38-48, vol. 6

[6] Andy Hunt, David Thomas, The Art of Enbugging, IEEE Software, January/February 2003, pp. 10-11, vol. 20.