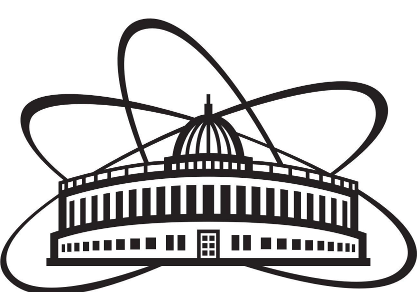
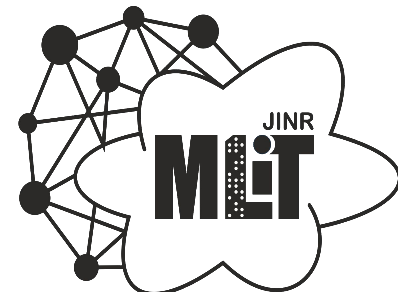




VIII SPD Collaboration meeting

Possible bottlenecks detection in SpdRoot code

Didorenko Aleksei
didorenko@jinr.ru

Voytishin Nikolay
nvoytish@jinr.ru

MLIT JINR

Relevance

Spin Physics Detector

International spin physics collaboration at the collider NICA

General information ▾ Collaboration ▾ Presentations and publications ▾ Setup ▾ Internal access ▾

General Information

- SPD CDR & TDR
- NEWS AND ANNOUNCEMENTS
- UPCOMING CONFERENCES
- CONTACTS
- USEFUL LINKS

Collaboration

- PARTICIPATING INSTITUTIONS
- EXECUTIVE BOARD
- TECHNICAL BOARD
- PUBLICATION COMMITTEE
- DOCUMENTS

SPD Presentations

SPD Software

SPD Software Wiki

Monte Carlo simulation, event reconstruction for both simulated and real data, data analysis and visualization are planned to be performed by an object oriented C++ toolkit SPDroot. It is based on the FairRoot framework initially developed for the FAIR experiments at GSI Darmstadt and partially compatible with MPDroot and BM@Nroot software used at MPD and BM@N, respectively.
The SPD detector description for Monte Carlo simulation is based on the ROOT geometry while transportation of secondary particles through material of the setup and simulation of detector response is provided by GEANT4 code. The standard multipurpose generators like Pythia6 and Pythia8 as well as specialised generators can be used for simulation of primary nucleon-nucleon collision.

- GIT Repository.

SpdRoot is a software package that is capable of performing Monte Carlo simulation, reconstruction, analysis and visualization of events.

It is stated that the reconstruction runs slower than expected.
The current issue is to detect bottlenecks in SpdRoot's source code and further improve the processing speed and efficiency of computing resources.

Purpose and tasks

Purpose of the work: to detect possible bottlenecks of the event reconstruction process in the SpdRoot’s source code.

Tasks:

- define a method for detecting bottlenecks
- find a tool to detect bottlenecks
- analyze the SpdRoot software package via found tool
- analyze the results

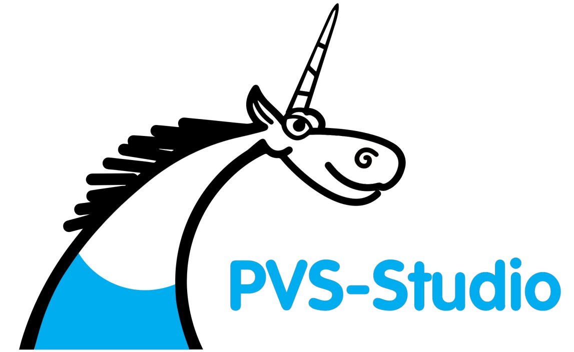
Technology stack

```
[alxdid@ncx104 ~]$ perf

 usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

 The most commonly used perf commands are:
   annotate        Read perf.data (created by perf record) and display annotated code
   archive         Create archive with object files with build-ids found in perf.data file
   bench           General framework for benchmark suites
   buildid-cache   Manage build-id cache.
   buildid-list    List the buildids in a perf.data file
   c2c             Shared Data C2C/HITM Analyzer.
   config          Get and set variables in a configuration file.
   data            Data file related processing
   diff            Read perf.data files and display the differential profile
   evlist          List the event names in a perf.data file
   ftrace          simple wrapper for kernel's ftrace functionality
   inject          Filter to augment the events stream with additional information
   kallsyms        Searches running kernel for symbols
   kmem            Tool to trace/measure kernel memory properties
   kvm             Tool to trace/measure kvm guest os
   list            List all symbolic event types
   lock            Analyze lock events
   mem             Profile memory accesses
   record          Run a command and record its profile into perf.data
   report          Read perf.data (created by perf record) and display the profile
   sched           Tool to trace/measure scheduler properties (latencies)
   script          Read perf.data (created by perf record) and display trace output
   stat            Run a command and gather performance counter statistics
   test            Runs sanity tests.
   timechart       Tool to visualize total system behavior during a workload
   top             System profiling tool.
   version         display the version of perf binary
   probe           Define new dynamic tracepoints
   trace           strace inspired tool

 See 'perf help COMMAND' for more information on a specific command.
```



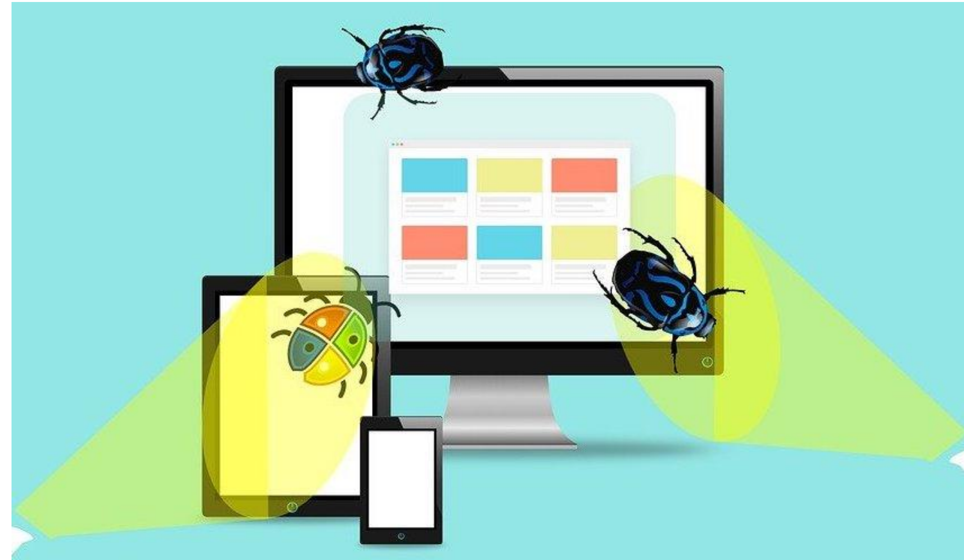
Profiling as a method for bottlenecks detection

Profiling is used to monitor the execution of a program to collect data on various aspects.

The purpose of profiling is to detect bottlenecks or areas where the program can be optimized to improve its efficiency and performance.

Profiling can be:

- static (analyzes the program code without executing it)
- dynamic (traces the program during its execution)



perf as a tool for analyzing software performance

perf is a dynamic profiling tool that is designed for Linux-based systems. Advantages:

- simple command line interface;
- rich functionality.

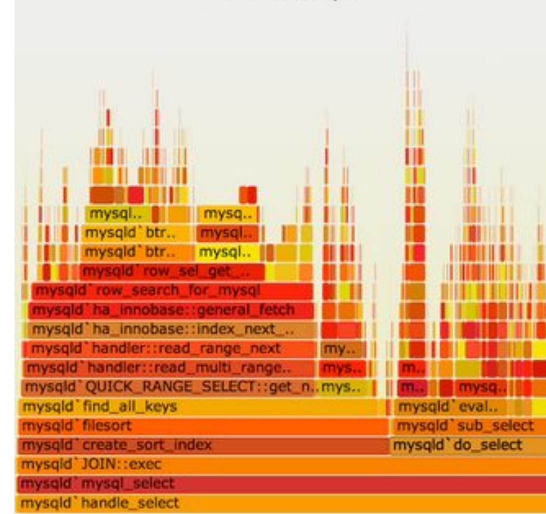
```
[alxdid@ncx104 ~]$ perf

 usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

 The most commonly used perf commands are:
   annotate        Read perf.data (created by perf record) and display annotated code
   archive         Create archive with object files with build-ids found in perf.data file
   bench           General framework for benchmark suites
   buildid-cache   Manage build-id cache.
   buildid-list    List the buildids in a perf.data file
   c2c             Shared Data C2C/HITM Analyzer.
   config          Get and set variables in a configuration file.
   data            Data file related processing
   diff            Read perf.data files and display the differential profile
   evlist          List the event names in a perf.data file
   ftrace          simple wrapper for kernel's ftrace functionality
   inject          Filter to augment the events stream with additional information
   kallsyms        Searches running kernel for symbols
   kmem            Tool to trace/measure kernel memory properties
   kvm             Tool to trace/measure kvm guest os
   list            List all symbolic event types
   lock            Analyze lock events
   mem             Profile memory accesses
   record          Run a command and record its profile into perf.data
   report          Read perf.data (created by perf record) and display the profile
   sched           Tool to trace/measure scheduler properties (latencies)
   script          Read perf.data (created by perf record) and display trace output
   stat            Run a command and gather performance counter statistics
   test            Runs sanity tests.
   timechart       Tool to visualize total system behavior during a workload
   top             System profiling tool.
   version         display the version of perf binary
   probe           Define new dynamic tracepoints
   trace           strace inspired tool

 See 'perf help COMMAND' for more information on a specific command.
```

One of the features of the perf tool is flamegraph, which visualizes hierarchical data, created to visualize traces of the profiled software stack to quickly and accurately identify the most common code paths.



Reconstruction startup parameters

The simulation and reconstruction were run using the example of the decay of a j-psi particle into two muons

NICA / spdroot

master

spdroot / macro / examples / jpsi-mumu

Updated jpsi-mumu example

Igor Denisenko authored 1 year ago

Name	Last commit
..	
analyze_jpsi.C	Updated jpsi-mumu example
fit_dimu.C	upadate 270321
reco.C	Updated jpsi-mumu example
run_all.sh	Updated jpsi-mumu example
simu.C	Updated jpsi-mumu example

The magnetic field is 1/8 of the total size

```
SpdFieldMap1_8 *MagField = new
SpdFieldMap1_8("full_map");
MagField->InitData("field_full1_8.bin");
SpdRegion *reg =
MagField->CreateFieldRegion("box");
reg->SetBoxRegion(-330, 330, -330, 330,
-386, 386); // (X,Y,Z)_(min,max), cm
run->SetField(MagField);
```

Reconstruction Flame Graph

FlameGraph displays the call of the reconstruction process functions.

- Each box - function;
- The width of the box shows the total time it was on-CPU;
- y-axis - stack depth;
- x-axis - sample population (sorted alphabetically);

colors are random

PVS-Studio as a tool for static code profiling

PVS-Studio is a static analyzer of C, C++, C# and Java code designed to facilitate the task of finding and fixing various kinds of errors: Improper understanding of function/class operation logic, Incorrect handling of the types, Misprints, Dead code, Copy-Paste, Uninitialized variables, Unused variables, Undefined/unspecified behavior, etc.

```
#Start analyze
pvs-studio-analyzer analyze -o /path/to/PVS-Studio.log \
                            -e /path/to/exclude-path \
                            -j<N>
#Get report
plog-converter -a GA:1,2 \
               -t json \
               -o /path/to/Analysis_Report.json \
               /path/to/PVS-Studio.log
```

Static code profiling results. JSON - report


```
{
   "code": "V678",
   "cwe": 688,
   "falseAlarm": false,
   "favorite": false,
   "level": 2,
   "message": "An object is used as an argument to its own method. Consider checking the first actual argument of the 'Transpose' function." ,
   "positions": [
     {
       "column": 1,
       "endColumn": 2147483647,
       "endLine": 966,
       "file": "/root/spdpvs/spdroot/external/GenFit2/trackReps/src/RKTrackRep.cc" ,
       "line": 966,
       "navigation": {
         "columns": 0,
         "currentLine": 1153117847,
         "nextLine": 1988692485,
         "previousLine": 1391284327
       }
     }
   ],
   "projects": [],
   "sastId": ""
}
```

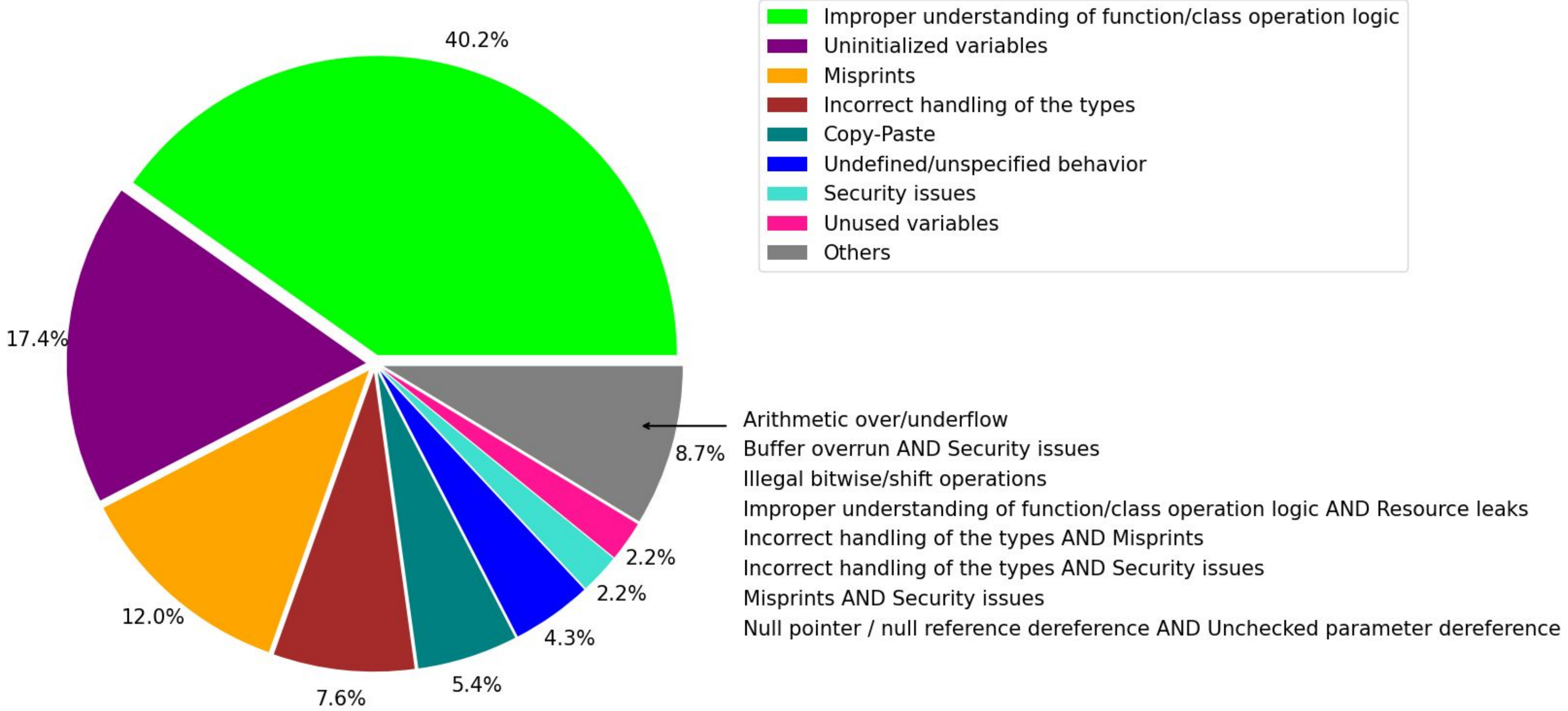
Static code profiling results. Errors

The pie chart shows errors fraction of each type of the total number of errors.

Sections – types of errors.

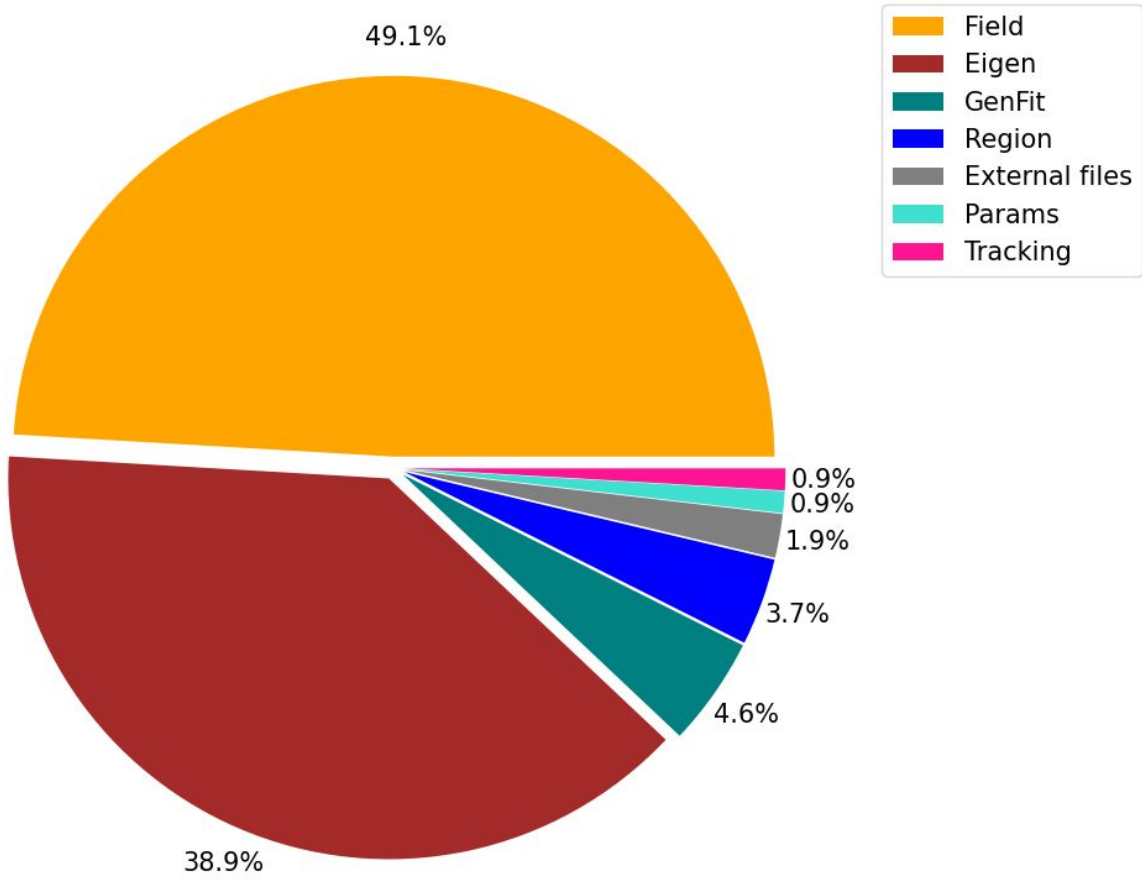
Percentages - the percentage of error type from the total number of errors.

Gray section contains types for which only one error was found (also errors that belong to several types at the same time).



Static code profiling results. Files

The pie chart shows SpdRoot code parts with errors fractions from the total number of files with errors
Sections – part of the SpdRoot code.
Percentages - the percentage of SpdRoot code part with errors from the total number of files errors.
Gray section contains fraction of external files.



Conclusion

- Dynamic profiling using perf did not give the desired results
- The static profiling method using PVS-Studio made it possible to detect errors as well as to identify sections of the SpdRoot code with a large number of errors

The results obtained can be useful to SpdRoot developers in improving the processing speed and efficiency of computing resources in the reconstruction process.

Thank you for your attention!