Performance Measurement Toolbox for GNU Radio

Marcus Müller, BSc.

marcus@hostalia.de

funkylab on freenode.org

Karlsruhe, Germany

Student at the Karlsruhe Institute of Technology

Google Summer of Code 2014

***Abstract*—GNU Radio has become one of the most popular frameworks for development of experimental wireless transceiver systems, especially in the research community. Research and Development however are in need of extensive metrics of such systems. A very common task, therefore, is variation of GNU Radio flow graph parameters and collecting values determined by running the system, including transceiver characteristics like bit error rate benchmarking, but there is rising need in the digital signal processing community to gather software performance data.**

This document proproses a Google Summer of Code project to create a unified, distributed and versatile tool to do benchmarking of signal processing applications as a whole and for performance analysis of their components.

I. Background

GNU Radio has been part of numerous research projects on wireless transmission systems, including development of modulation schemes, medium access control strategies and is in the process of getting a increasingly comprehensive channel coding framework.

With the creation of `gr-benchmark`, steps were taken to profile the execution of GNU Radio applications as well as components on different computing platforms. However, this is limited to benchmarks run on the local computer and to extracting key data for analysis and potential upload to a central server[1].

For the development of complex transceiver systems, this covers but an aspect of the overall desirable benchmarking tools:

One typical benchmark is the performance of such a system is the bit error rate (BER) curve over varying SNR conditions. While there are plenty options to gather data from GNU Radio flow graphs and generate such statistics from them, there is a distinct lack of tools that enables researchers and developers to run extensive benchmarks in an easy, reproducible, and comfortably analyzable ways.

Furthermore, benchmarking a complex simulation can take quite some computation time. It is highly desirable that benchmarks can be automatically distributed to remote systems, the results being gathered on a central node. This has to be done by a system that *guarantees* that results are properly labeled, archived, the parameters documented and stored in a way that enables visualization as much as sharing and analysis using standard tools.

The main objective of `gr-benchmark` is to profile the computational performance of different implementation of algorithms and mathematical base operation, especially for the VOLK library[2]; it implements a method to upload profiling data to the central http://stats.gnuradio.org server for analysis by the VOLK community. Obviously, it is desirable that when developing performance-critical code one is able to test performance on several machines while coding is still in progress, and not just when code is *out in the wild*; that's basically the same reason GNU Radio encourages developers to use the tightly integrated unit testing tools when developing in and out of tree modules.

This brings up the need for a centralized dispatcher mechanism that distributes the execution of benchmarks to different machines according to rules that either govern the execution of each benchmark on each machine for computational profiling and testing purposes or the minimization of the execution time of a benchmark suite for processing performance benchmarking.

II. Proposed Project

A. Extension of `gr-benchmark`

Application performance measurement: So far, `gr-benchmark` is tailored to the need of the VOLK community to benchmark specific implementations of mathematical routines on different architectures.

This leaves room for unforeseen behaviour: Running a highly optimized VOLK kernel in a benchmark potentially provides the kernel with CPU cores for nearly exclusive use; realistically, these kernels are part of larger GNU Radio applications. This leads to competition for CPU cores, and even more importantly, for caches. If a kernel works well on its own, it might actually profit greatly from caching mechanisms. If, however, the complete system is to process a larger number of sample buffers, cache misses will occur far more often. To measure the performance of kernels, it therefore seems reasonable to compare the time spent on the kernel with the total computation time of an application employing that respective kernel.

Signal Processing Focus: As a very common task is the simulation of a transceiver system under different environments (SNR, Frequency Offset, Interference, Timing misbehavior). `gr-benchmark` should be expanded to accommodate the FEC API as well as to be configured to run simulations with a specified set of simulation parameters.

I anticipate that GNU Radio users will want to have an fail-safe, easy way to get the desired data out of the flow graph. A set of sinks to let users specify the data to be collected will be created; this, aside from the obvious BER, should also include estimators for SNR, residual synchronization errors, etc.

B. Dispatcher Infrastructure

An infrastructure to remotely execute benchmarks, especially flow graphs, must be supplied to fulfill the need for reproducible, centralized collected distributed benchmarks.

Technologically, existing methods for the coordination should be employed. As a means to distribute the benchmark programs, to ensure compatible GNU Radio versions on all machines and to start the remote benchmarking agents, `ssh` offers a well-established framework.

To queue individual benchmarks on the remote machines, a remote procedure call (RPC) library should be used. Although `gr-controlport` already employs `ICE` for RPC, which itself is networkable, defining interfaces in shape of slices implies a considerable development overhead for the integration of quickly changing benchmarks.

`ØMQ` [3] on the other hand has been successfully employed in GNU Radio applications[4], and already offers the optimal architecture in an easy-to-use framework (fig. 1).

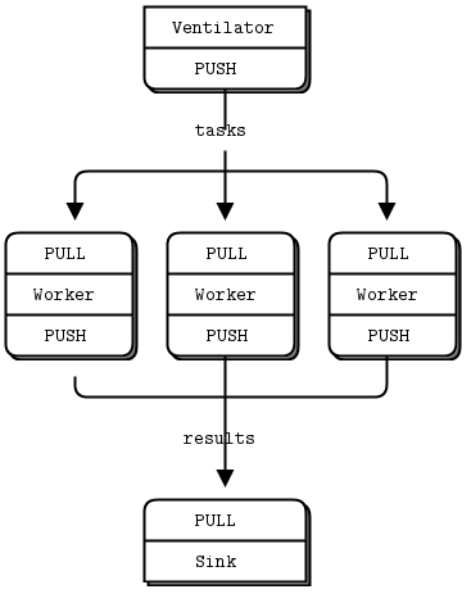

Fig. 1. `ØMQ`-based push/pull based workload distribution. From: http://zguide.zeromq.org/page:all#Divide-and-Conquer

Since there are several attractive alternatives to `ØMQ`, such as the more python-centered and deployment focused `execnet`[5], choosing the right framework is not inherently trivial, and the decision should be one of the initial tasks during framework development.

The dispatcher architecture must be somewhat failure tolerant: Benchmarking results should be stored locally on the nodes, so that in case of infrastructure failure (network outage, dispatcher crash, ...) results are not lost but can be collected later.

Execution of Benchmarking Plans and Storage of Results: The overall benchmark, specifying the flow graphs, kernels or generally programs to be run, should be stored in *plans*. The dispatcher should be able to load them from permanent storage.

Results of Benchmarks should be stored in a relational database. For smaller plans, an embedded database like `Sqlite3` is sufficient. Using one of the numerous `python` RDBMS abstraction interfaces, seamless integration into high-performance database systems should be possible.

Signal Processing Benchmark: For computationally intensive benchmarks, distribution of benchmarking subtasks across multiple systems is always desirable. In a lab environment, often a rather heterogeneous network of computers is available to execute the benchmark (fig. 2).

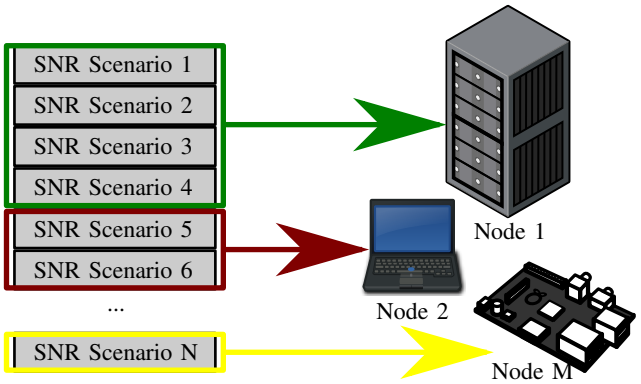

Fig. 2. Benchmark distribution to achieve minimal computation time

A dispatcher user interface should enable users to run their benchmarks on the different machines, and enable them to gather results without explicitely assigning benchmark to the different nodes.

In this scenario, the *pool manager* should have means to check the state and availability of assigned *nodes*, remove them from or add them to the pool, monitor the individual node performance and allocate workload accordingly to minimize total computation time.

Computational Performance Benchmark: To test e.g. optimized algorithm on different machine types, or to gather statistical properties (average, median, variance) of performance, benchmarks must be run on a number of different machines simultaneously (fig. 3).

C. Benchmark Design Suite

The previous sections have shown that, depending on the benchmark type, demands for benchmark scheduling are completely different. To account for the complexity involved, an easy-to-use graphical interface for the planning of benchmark should be devised.

Integration into `gnuradio-companion`: The `grc` offers users the possibility to define various variables that can be used to parameterize blocks.

I plan *Benchmark* as a new *generate option* for `grc`. This will enable the user to specify *start*, *stop*, and a number of *steps* of each parameter in a dedicated input window listing

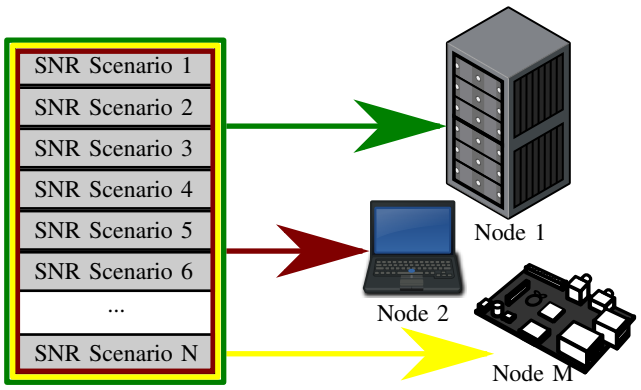

Fig. 3. Benchmark distribution to all nodes, to run every benchmark on every platform

all user-specified variables, as well as the desired number of *repetitions* per run. These settings along with the flow graph should be exported to a *plan* that can be dispatched. Since graphical sinks do not make much sense in remotely executed benchmarks, integration of the GUI toolkit environments into that generate option is currently not planned.

The generated benchmarks should be exported in dispatcher-loadable plans.

D. Benchmark Analysis

Publication is a necessity for scientific progress to be available to every researcher. To ease the generation of visuals from the gathered benchmark data, the toolbox will come with scripts that analyze the stored benchmarking results, extract the desired data, and export it to comma-separated-value files, plot it directly using `matplotlib` ready for LaTeX import, or publish it (in the case of performance data) to an online statistic website.

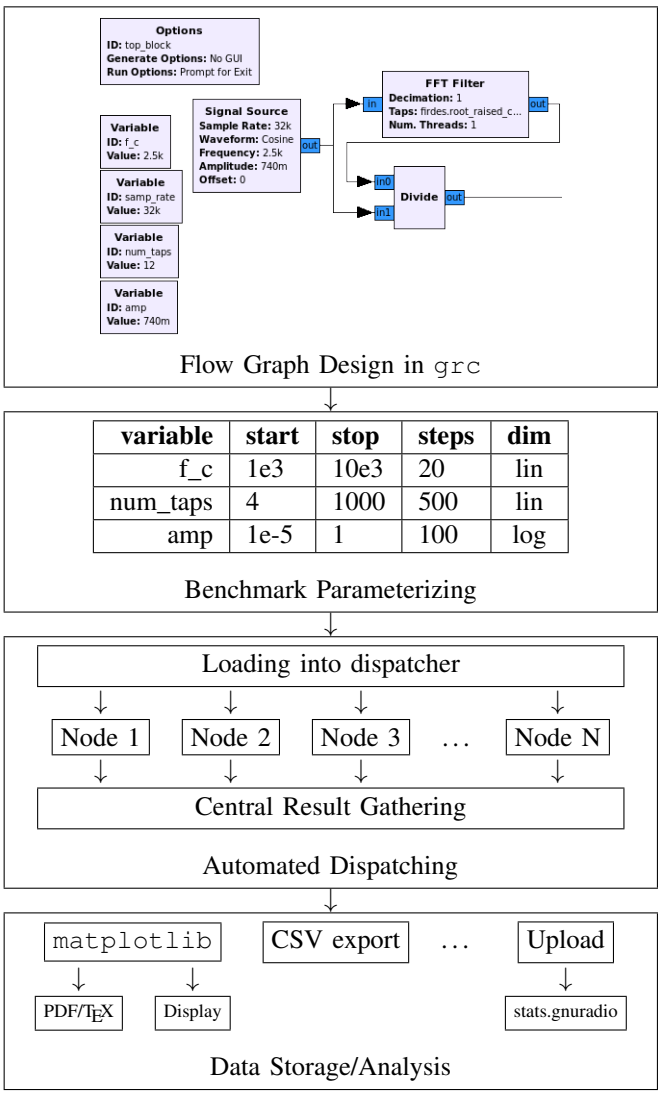

Fig. 4. Typical Benchmarking workflow

III. Deliverables

A. Extension of `gr-benchmark`

- Integration of signal processing performance Measurements
- `gr-benchmark`-enabled blocks to help users specify the relevant data
- relative runtime measurements under load
- Focus on statistical properties (average, variance etc.)

B. Dispatcher Infrastructure

- `SSH` infrastructure
  - automatic login via public key authentication
  - pool management
  - starting of execution agents
- Pool manager
  - add, remove nodes
  - start agents from CLI
- Execution agents
  - RPC infrastructure
  - local data storage
  - transmission format (XML?)
- Dispatcher
  - plan loading
  - task distribution
  - result gathering
  - result storage (database interface)
  - fault analysis
  - GUI
- Benchmark Design Suite
  - `grc` integration
  - parameter definition
  - plan Saving
- Benchmark Analysis
  - data extraction
  - export facilities (CSV, `matplotlib`)
  - upload (to http://stats.gnuradio.org)

IV. SCHEDULE

This schedule is rather detailed; I honestly expect it to change in the coordination period.

Currently I'm employed on a 9 hours per week basis, but the schedule should leave me with enough flexibility to do Google Summer of Code and that in parallel. Further commitments can be reduced, even eliminated, to ensure reaching of weekly goals.

This schedule already includes the long weekends I intend to take.

April 22 May 18	**Coordination period:** • refining a wish list for `gr-benchmark` features • defining data formats, libraries used • defining test cases • setup of a VM for distributed tests • setup of github repo • fill github issue tracker with feature requests, use from thereon • exchange with experts (on `grc`, `gr-controlport`, ...) active communication on IRC announcement and RFC on discuss-gnuradio
May 19	**Start of work period:** constant updates on github active communication via IRC, mailing list
May 19 May 24	`gr-benchmark` extension I • whole-application benchmarks • data extraction blocks
May 26 May 31	Infrastructure I • choose right RPC framework (ØMQ? `execnet`? ...) • benchmark dispatcher • benchmark running agent
June 2 June 5	Integration I • dispatcher plan loading • plan generation GUI • `grc` generate option
June 10 June 14	Infrastructure II • wrapping tasks & results (XML?) • result gathering • result storage
June 17 June 21	Infrastructure III • `SSH` infrastructure – public key storage – start of agent – checking of GNU Radio version
June 23 June 27	**Mid-term evaluation submission**
June 30 July 9	Work on supplied feedback
July 10 July 12	Integration II • local setup script – generation of system user – public key generation – `pybombs` recipe • update of VM
July 14 July 19	Data Analysis I • database data extraction • visualisation of statistical properties • export
July 21 July 26	`gr-benchmark` extension II • refine to store data directly to local Infrastructure IV • delayed & failure tolerant gathering of results • upload facilities
July 28 August 2	**Extensive testing & Bugfixes** • deployment on different platforms • VM deployment to amazon EC2 (micro) • GNU Radio QA integration
August 4 August 9	Integration IV • Dispatcher UI – Integration into `grc` • Visualization UI
August 11 August 16	Documentation
August 18 August 22	**Finalizing** (code cleanup, repo cleanup)

Marcus Müller first came into contact with GNU Radio in 2010, when he was part of a team project at the Communication Engineering Lab (formerly: Institut für Nachrichtentechnik) that yielded a MUSIC and ESPRIT spectrum estimators[6], which were later integrated in `gr-specest`[7].

Since then, he did different GNU Radio projects, amongst which there is his bachelor thesis on OFDM-based Radar[8] and a demonstrator for cognitive radio sensing[9].

He's been an active participant on the gnuradio-discuss as well as on the usrp-users mailing list, and tends to idle around in `#gnuradio`.

As someone who has been employed as student research assistant, he knows how much time and effort the generation of statistically sound benchmarks costs.

Since programming of digital signal processing algorithms is one of his major interests, performance analysis and optimization are attractive occupations to him.

Right now he is working on his Master's degree in electrical engineering.

REFERENCES

[1] Timothy J. O'Shea and Thomas W. Rondeau , "A Universal GNU Radio Performance Benchmarking Suite," in *Proceedings of the Eighth Karlsruhe Workshop on Software Radio*, 2014.

[2] Vector optimized library of kernels. [Online]. Available: http://gnuradio.org/redmine/projects/gnuradio/wiki/Volk

[3] ∅MQ: Code connected. [Online]. Available: http://zeromq.org

[4] gr-airmodes: A GNU Radio Framework for reception of ADB-S signals. [Online]. Available: https://github.com/bistromath/gr-air-modes

[5] execnet: Distributed python deployment and communication. [Online]. Available: http://codespeak.net/execnet/

[6] S. Ehrhard, M. Fischer, M. Fuhr, M. Mouazzen, M. Müller, and M. L. Schulz, "Spektralschätzung mit MUSIC und ESPRIT," Karlsruhe Institut of Technology, team project, 2011. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/1678008

[7] GNU Radio spectrum estimation toolbox. [Online]. Available: https://github.com/kit-cel/gr-specest

[8] M. Müller, "Software Radio-basierte Implementierung von OFDM-Radar Algorithmen," Bachelor's thesis, Karlsruhe Institut of Technology, 2012.

[9] A. Kaushik, M. Mueller, and F. K. Jondral, "Cognitive relay: Detecting spectrum holes in a dynamic scenario," in *Proceedings of the Tenth International Symposium on Wireless Communication Systems*. VDE, 2013. [Online]. Available: http://www.cel.kit.edu/download/kaushik_ISWCS_2013.pdf