Inspecting GNU Radio Applications with ControlPort and Performance Counters

Thomas W. Rondeau
University of Pennsylvania
Philadelphia, PA 19104, USA
tom@trondeau.com

Timothy O'Shea
University of Maryland
College Park, MD, 20742
oshea@umd.edu

Nathan Goergen
University of Maryland
College Park, MD, 20742
goergen@umd.edu

ABSTRACT

Due to differences in the operating system and the effects of sample rate on the computational load of a software radio, we have historically had a difficult time understanding the performance boundaries of software radio applications. This problem further leads to difficulties in debugging, optimization, and profiling analysis of both software radio frameworks and applications.

This paper introduces a new tool developed for GNU Radio that starts to solve these problems. Called *Performance Counters*, GNU Radio now has an inbuilt ability to measure its performance for offline optimization as well as real-time behavioral analysis and adaptation. The Performance Counters are designed such that a GNU Radio application can directly sample them or access them through the use of *ControlPort*, another new tool that enables remote interaction with GNU Radio. We show in this paper some of the tools we have developed around ControlPort and the Performance Counters that will help us better understand GNU Radio's performance and capabilities as well as lead to better on-line adaptation of radios.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Design, Experimentation, Measurement, Performance

Keywords

GNU Radio, Profiling, Signal processing, Software radio

1. INTRODUCTION

Performance analysis of software radios based on general purpose processor (GPP) has been consistently problematic and under-analyzed. Unlike strongly clocked systems like FPGAs where system performance metrics are well understood, like the idea that an FPGA image must meet timing, GPP-based applications running on modern operating systems (OS) have a much more amorphous relationship with performance and performance metrics. Commonly, the practice for what kinds of waveforms a software system can handle is just to ask the question "does it work?" If it does not, then we try to identify the areas of heavy processing and optimize them. But finding these areas is itself a whole new problem.

GNU Radio [1] has never had a comprehensive response to the above problem. We recently introduced new tools that will help solve these issues and address other similar problems related to performance, profiling, and life cycle monitoring.

In this paper, we present the new concept in GNU Radio called *Performance Counters* along with an architectural add-on known as *ControlPort* that is one of the most effective ways of using and interacting with the Performance Counters. Specifically, the Performance Counters are a set of statistics built into the GNU Radio scheduler that keep track of values of interest for every block operating in a GNU Radio flowgraph. We cover these values more in Section 4 while we introduce ControlPort beforehand in Section 3. First, we discuss in some more detail the particular problem we face.

2. BACKGROUND

Operating systems such as Linux have, especially with multi-threaded applications, unknown execution times and limits. For software radio where we require a system to support, often in full-duplex mode, specific data rates of a waveform, the ambiguity of the operating environment, software interrupts, and external process handling makes guarantees difficult and absolute certainty impossible. Furthermore, we have a difficult task in even understanding the limits of a system and whether our software radio application can fit within any established limits we try and define.

To put it another way, the computational cost of a software radio processing chain for a waveform is dependent on the waveform's symbol rate, the algorithms required to process the waveform, and the implementation of those algorithms. In an ASIC or even FPGA, we have structural guarantees that the implementation of the algorithms acts within the time period required by the waveform's properties. But the shared scheduling of an OS like Linux makes it such that we cannot specifically determine the amount of

[1] `gnuradio.org`

SRIF'13, August 12, 2013, Hong Kong, China.

computational overhead required. GNU Radio has an even further problem in that it is heavily cross-platform between GPP types of all kinds and generations. We have no metrics to understand how one waveform operating flawlessly on a particular processor will behave when moving to another processor of a different family (e.g., Intel x86 to ARM) or between generations (Intel Core i7 to a Core2Duo).

In response to this, we want a profiling system that enables us to do both on-line adaptation of GNU Radio applications as well as provide some insight into areas of optimization. Going back to the question of performance of a particular waveform, if the processing chain is found to be too slow for the processing requirements, we might be able to do better by optimizing certain blocks and algorithms. We would like to find out which blocks in particular are causing problems and so be able to better focus our optimization efforts.

We are becoming increasingly interested in extending the use of GNU Radio onto many-core systems. This problem poses the challenge of efficiently mapping block threads and memory resources to physical resources, which may be significantly more difficult for large applications. Knowing the computational requirements of each block as well as the data-flow topology could lead us to better processing element assignment, such as allowing multiple inexpensive blocks to share a core while another, more complex block uses a dedicated core and thereby reducing memory bandwidth requirements.

Despite all of these issues discussed above, we have not had a good, comprehensive, and easy-to-use ability to study the processing capabilities of a GNU Radio applications. There are a handful of tools that we have employed in the past to help in our understanding but these can be platform-specific, difficult to use, and limited in scope.

One set of tools that we have made use of for determining the cost of the algorithms in our application are profiling tools. Such tools like Oprofile [2] and Valgrind's cachegrind [3] are useful to an extent. Both tools, however, can be hard to use and the results difficult to analyze, especially when looking at large applications. Oprofile is Linux-specific, and Valgrind runs the software under simulation. In both cases, a program is run for a set amount of time and the data then stored for later analysis. This model has allowed us to identify some optimization points in our code, but is not completely helpful from a large-picture perspective. Not to diminish the usefulness of these tools, once an algorithm or block is identified as needing optimization, these tools can be particularly effective in finding where optimization efforts should focus as they can provide a line-by-line analysis in the source code of activity.

The Performance Counters, on the other hand, are instrumented directly into GNU Radio, which makes them easily usable in both on-line and off-line analysis. The Performance Counters are also developed specifically for software radio issues, so they cover metrics that tell us about the signal processing and data flow behavior in an application.

2.1 Previous Work

There has been other work done that addresses these problems in software radio. Two such projects are ALOE [4] and Surfer [1]. These projects were built around the desire to investigate and research software radio issues, so performance understanding is key to their original architecture.

ALOE is designed around heterogeneous, multi-processor platforms [4] as well as grid-based or cloud computing architectures [5]. To accomplish this, they structure their code around time slots and require operations to happen on timed boundaries. This both allows them a better understanding of the processing requirements but also locks them into certain structural requirements when building their signal processing blocks. GNU Radio optimizes throughput by always pushing as much data as possible through the system, which allows fewer design constraints but also fewer timing guarantees.

Surfer is another research-oriented software radio architecture [1]. It is also centered on the concept of heterogeneous multi-processor platforms and uses in-built runtime statistics to understand the processing capabilities and requirements to enable runtime reconfiguration. One of Surfer's purposes is to use a supervisor to collect runtime statistics such as throughput, latency, processor load, etc. and make decisions to dynamically adjust the radio [3].

Both ALOE and Surfer have informed the use of statistics of software radio, and Performance Counters are the result of developing similar ideas that fit the architecture and design goals of GNU Radio. As a research field, we still have a lot to learn about what these statistics tell us about the radio's operation and how to best use them to inform our use and runtime behavior of a radio. Some of these thoughts are discussed in the Surfer work as found in [2] and developed further with some interesting plotting techniques in [1].

3. CONTROLPORT

ControlPort is a new tool that creates an integrated remote procedure call interface to GNU Radio. Any block or part of GNU Radio can now register interfaces with ControlPort, which then allows a ControlPort client to interact with the GNU Radio application through any one of these interfaces. The general form of these interfaces is a *set* and/or *get* function to adjust or query the state of a GNU Radio block's parameter. For instance, a phase lock loop can have an interface to set the loop bandwidth so as to optimize the acquisition of different signals. Likewise, an FEC decoder or signal recovery block might expose an error measurement so that an operator could remotely test the current fidelity of a received signal.

Figure 1 shows a simple illustration of ControlPort. GNU Radio applications are distributed but connected to the Internet through the host platform. ControlPort creates an endpoint at the local IP on some port, which then allows ControlPort clients to open a TCP connection and start interacting with the interfaces in the application.

GNU Radio's ControlPort sits on top of the Internet Communication Engine (ICE) library [4]. We use this to provide us with a library of features for setting up and tearing down connections, stability, security, exception handling, and efficiency.

ControlPort is very light-weight from a processing perspective. When it has no active requests, it is in an idle blocking mode and takes no extra processing time. Likewise, it dispatches the remote requests in threads separate from the signal processing. For a ControlPort probes, it

[2] `oprofile.sourceforge.net`

[3] `www.valgrind.org`

[4] `www.zeroc.com`

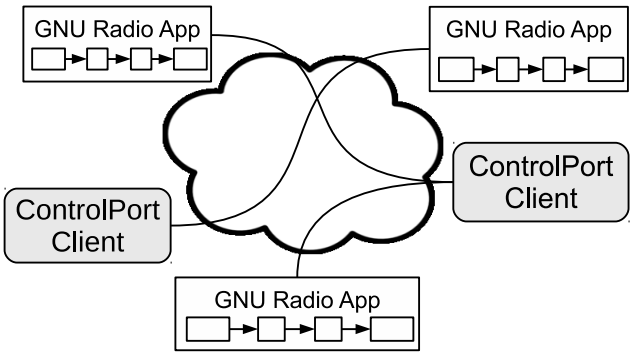

Figure 1: Illustration of ControlPort with three GNU Radio applications and two ControlPort clients communicating over a TCP connection through the Internet.

simply pokes a memory location, which results in minimal impact to the dataflow.

We introduce ControlPort in this paper because it is a vital underpinning of our Performance Counter interface and the performance monitoring tools. We will link these together in the next section after introducing the Performance Counters.

4. PERFORMANCE COUNTERS

GNU Radio's *Performance Counters* are a way of generating performance measurements for each block running in a GNU Radio flowgraph. These counters keep track of various states of each block that can then be used for analyzing the performance and behavior of a running flowgraph. Currently, there are five identified performance counters:

noutput_items number of items the block can produce.

nproduced the number of items the block produced.

input_buffers_full % of how full each input buffer is.

ouptut_buffers_full % of how full each output buffer is.

work_time number of CPU ticks during the call to general_work().

When calculating the performance counters, each block keeps track of the most recent value (*i.e.*, the instantaneous value) as well as the running average and running variance of the counter. We can retrieve all three values programmatically using the function calls on the blocks themselves, described in the GNU Radio manual.

We want to point out that the performance counters mentioned in this paper are part of our initial work into the area. We expect to both add more performance counters and potentially alter the exact definitions of these five already described. This paper represents our start of the investigation into this subject.

Although we expose the performance counters through direct function calls to a GNU Radio block, we designed the performance counters to be usable in a variety of ways. We are especially interested in sampling the performance of a currently running flowgraph as well as being able to measure the performance on embedded devices where direct access and visualization tools are not readily available or practical. We therefore export all performance counters for all blocks automatically over ControlPort. We can now use ControlPort to sample the performance counters at any time during the life of the flowgraph.

To improve how we access the performance counters, we created a tool designed to read the counters and display them to help visually represent the state of the flowgraph. This tool is called **gr-perf-monitorx** and comes included with GNU Radio as of version 3.7. The tool is a QT-based application that uses Python modules Scipy[5], Matplotlib[6], and NetworkX[7].

When launched, **gr-perf-monitorx** pulls a representation of the GNU Radio flowgraph over ControlPort and constructs a graph of nodes, which represent the GNU Radio blocks, and edges, which represent the buffers between blocks. The size of each node is proportional to the performance counter "work_time" while the width and darkness of an edge is proportional to the performance counter "ouptut_buffers_full". It periodically polls the flowgraph using ControlPort to update the graph's nodes and edges. This tool gives us an immediate visualization of how each block is performing in the running flowgraph allowing for rapid identification of any bottlenecks or other performance limiters.

The **gr-perf-monitorx** also allows us to open tables and bar graph plots for both the work time and output buffer states so that we can directly look at the actual numbers or visual comparison of each block's performance.

5. USING PERFORMANCE COUNTERS

Here we show how the performance counters and the monitor application work. The flowgraph is a simple application that uses two different resamplers to change the sampling rate of a signal received through a USRP N210 [8]. Both the fractional interpolator and polyphase filterbank arbitrary resampler adjust the sample rate of the incoming signal by some real number amount. The signal being received by the USRP is an OFDM waveform generated from another USRP.

Figure 2 shows the flowgraph used to capture the signal. It uses no graphical user interfaces (GUIs) itself but sends the signal through two different resampler block and into ControlPort probes. These probes are specifically designed to work with ControlPort so that we can capture and analyze the signal remotely.

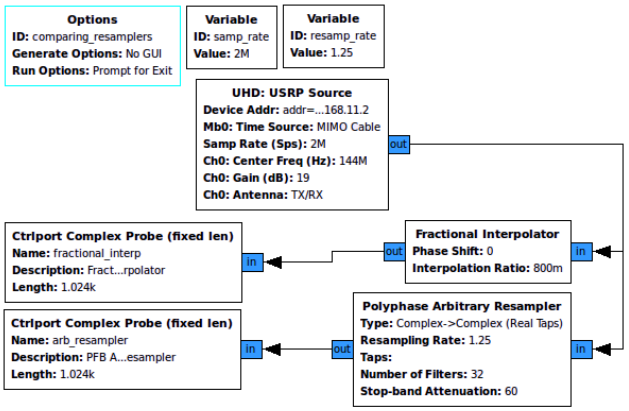

Figure 2: Flowgraph receiving a signal from a USRP N210 and resampling using two different algorithms.

[5] www.scipy.org
[6] matplotlib.org
[7] netowrkx.github.com
[8] ettus.com

On another computer in the lab network, we use **gr-perf-monitorx** to visualize the performance characteristics of this GNU Radio application. Figure 3 shows the performance monitor tool. As described, it reconstructs a local representation of the flowgraph over ControlPort. Because the node size is proportional to the work time and the edge width and darkness proportional to the buffer fullness, the graph gives us a very easy visualization tool to see the relative performance of each block, which will help us in understanding where any bottlenecks are or where to focus our time in optimizing blocks.

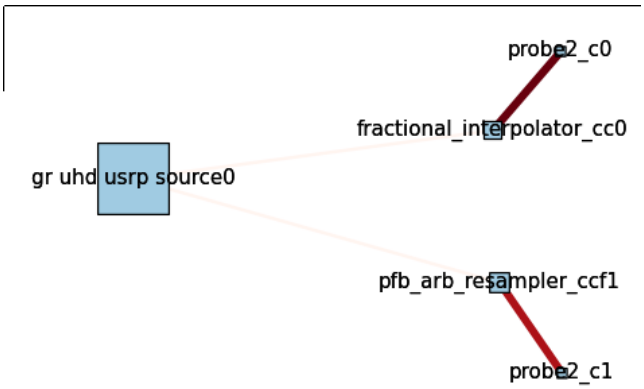

Figure 3: Reconstructed flowgraph over ControlPort to represent the performance of the resamplers.

The performance monitor tool's graph of Figure 3 shows us a couple of things immediately. First, the UHD source block that reads in samples from the USRP device is where most of the time is spent in this application. We know this from the size of the block. However, we can also see that the buffers from the UHD source to either resampler are also nearly empty as they are a very light pink. This means that all samples produced by the UHD source are almost immediately consumed by the resamplers. The resamplers then, are next in the amount of work time spent in them while the ControlPort probes are computationally light.

It should be noted that this measurement is currently done using the system real time clock and is not a representation of the user and system time of the thread, meaning that actual time spend in the UHD source work function, including long blocking read calls for data and other thread preemption, yields, or context switches are included in the measurement. Alternatively, the user time of the CPU time clock may be used to inspect consumers of compute time while disregarding the impact of necessary blocking read or wait calls.

While the sizes of the nodes and edges give some indication of what is going on, it can be hard to distinguish exact values just from graph. To that end, we also have the ability to bring up both a table and bar graph format for the work time and the buffer fullness amount. We show the runtime table and bar graph in Figure 4 to get a better understanding of the differences between the resamplers. All of the numbers are normalized so that they sum to 1.

In this case, again, the majority of the time is obviously spent in the UHD source block. What we can also see is that the PFB resampler is slightly more expensive than the fractional resampler. This is really useful to know. One of the benefits of the PFB resampler is that it incorporates a

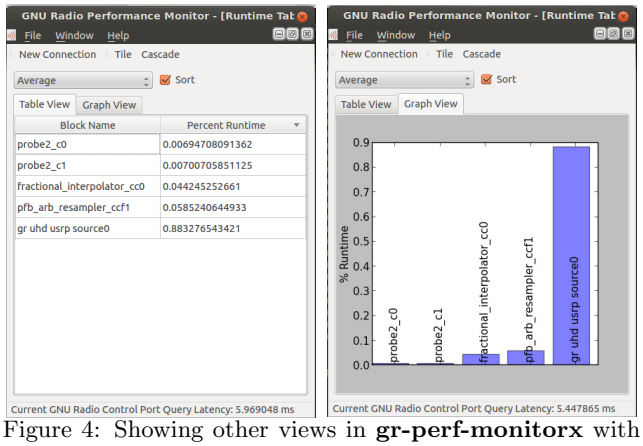

Figure 4: Showing other views in **gr-perf-monitorx** with a) a table list of the normalized values and b) the same values compared in a bar graph.

filter that can be hand-tuned, so we can get the benefit of two operations at once. However, if we do not need to do any further filtering, we can save some computational overhead by using the fractional interpolator block.

Notice in the table and bar graphs that there is a options box that says "Average." The Performance Counters keep track of the latest, or instantaneous, value of each counter as well as the average and variance. All forms of the displays in the **gr-perf-monitorx** can be set to display any version of the counters.

5.1 Testing Simulations

We can similarly analyze complex applications, even under simulation. Figure 5 shows a fairly complicated flowgraph that generates random samples modulated as QPSK, passes them through a channel model, and performance timing and phase recovery. It has three ControlPort probes to monitor the state of the signal at the output of the channel model, the output of the timing synchronizer, and the output of the phase synchronizer.

We note that the "PSK Mod" block and "Channel Model" block are actually hierarchical blocks, which means they are made up of other GNU Radio blocks and presented together as a convenient grouping. When we pull the flow graph over ControlPort for looking at the Performance Counters, we ignore the hierarchy and instead get a full representation of then entire flowgraph.

The full flowgraph is shown in the performance monitor tool capture in Figure 6. There are are many more components here than in the original flowgraph, which is what we want in order to analyze the behavior of all blocks and buffers for any particular bottlenecks.

The performance monitor shows that the complicated algorithms, as expected, take most of the processing time, including the arbitrary resampler and timing and phase synchronizing loops. We also see that the noise source that is part of the channel model is quite heavy on the processing requirements. All of this is information we can use when looking to optimized our system's performance.

6. CONCLUSION

With its widespread use and continuing development of complex waveforms, GNU Radio has needed a more robust

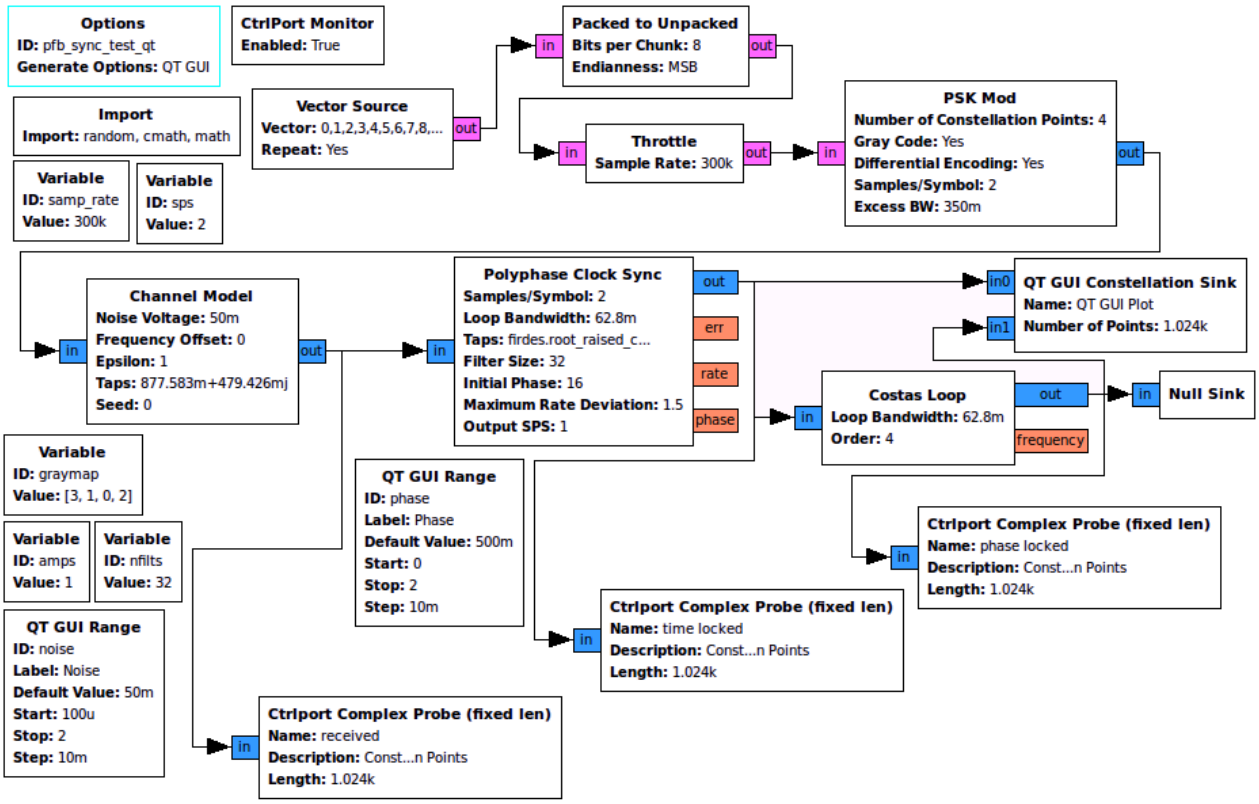

Figure 5: A flowgraph simulating a QPSK signal through a channel and basic synchronization with ControlPort probes.

method of measuring and monitoring performance. Other research-oriented platforms have already started to explore this space. GNU Radio has recently started to research this area through the concept of Performance Counters. Through the use of Performance Counters over ControlPort, we show here a new and powerful way of interacting and therefore controlling radio applications based on built-in performance statistics. We still have a lot to learn about what these values tell us about the radio's performance, resource utilization, and even areas that require further optimization. These Performance Counters should change how we analyze and interact with GNU Radio applications, but we are still at the early stages and are continuing to develop our understanding of this space and its impact on GNU Radio.

7. REFERENCES

[1] M. L. Dickens. *Surfer: Any-Core Software Defined Radio*. PhD thesis, University of Notre Dame, Notre Dame, IN, Apr. 2012.

[2] M. L. Dickens, B. P. Dunn, and J. N. Laneman. Thresholding for optimal data processing in a software defined radio kernel. *Proc. Karlsruhe Workshop on Software Radios (WSR)*, Mar. 2010.

[3] M. L. Dickens, J. N. Laneman, and B. P. Dunn. Seamless dynamic runtime reconfiguration in a software-defined radio. *Proc. SDR WInnComm Europe*, Jun. 2011.

[4] I. Gomez, V. Marojevic, and A. Gelonch. Aloe: An open-source SDR execution environment with cognitive computing resource management capabilities. *IEEE Communications Magazine*, 49(9):76–83, Sep. 2011.

[5] I. Gomez Miguelez, V. Marojevic, and A. Gelonch Bosch. Resource management for software-defined radio clouds. *IEEE Micro*, 32(1):44–53, Jan.-Feb. 2012.

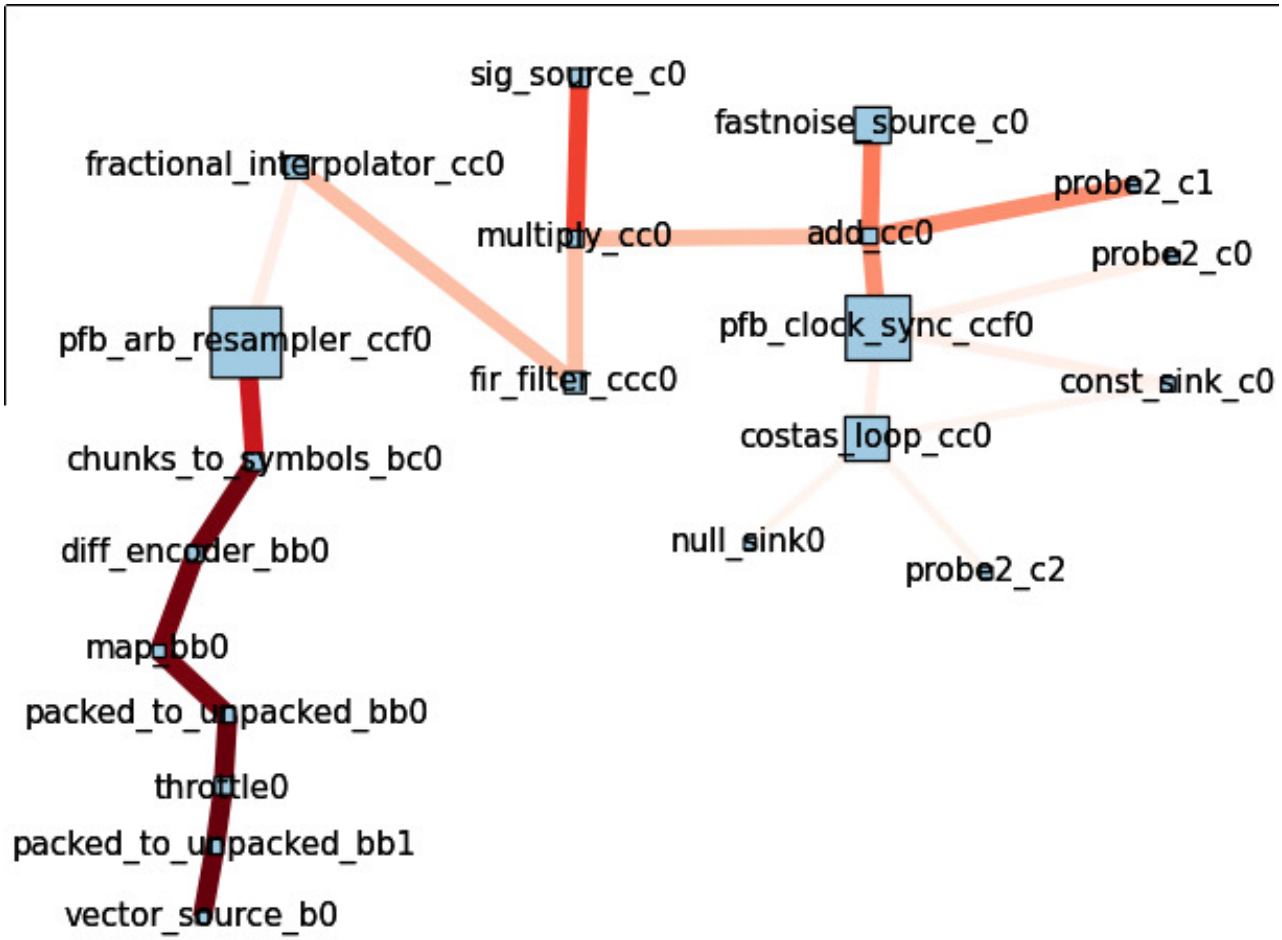

Figure 6: Performance monitor tool for Figure 5.