Benchmarking and Profiling the GNU Radio Scheduler

Bastian Bloessl MAIL@BASTIBL.NET
Secure Mobile Networking Lab, TU Darmstadt, Mornewegstr. 32, 64293 Darmstadt, Germany

Marcus Müller MUELLER@KIT.EDU
Communications Engineering Lab, Karlsruhe Institute of Technology, Kreuzstr. 11, 76133 Karlsruhe, Germany

Matthias Hollick MHOLLICK@SEEMOO.TU-DARMSTADT.DE
Secure Mobile Networking Lab, TU Darmstadt, Mornewegstr. 32, 64293 Darmstadt, Germany

Abstract

From a technical perspective, GNU Radio has two main assets: its comprehensive block library of optimized, state-of-the-art signal processing algorithms and its runtime environment. The latter manages the data flow and turns GNU Radio in a real-time signal processing framework. In contrast to the block library, where it is easy to replace blocks with more efficient implementations, the runtime grew organically, which resulted in a complex system that is hard to maintain. At the same time, there are concerns about its performance. To understand the current implementation and explore opportunities for future improvements, we provide benchmarking and profiling results. We, furthermore, compare the performance of GNU Radio's default with a manually optimized configuration to show the potential of a more advanced scheduler.

1. Introduction

Many current and forthcoming wireless technologies have high bandwidth demands and tough latency constraints. This is a challenge for Software Defined Radio (SDR), in particular for General Purpose Processor (GPP)-based platforms like GNU Radio (Rondeau, 2015), where signal processing is implemented on a normal PC. To meet the demands of new technologies, GNU Radio focused on optimizing individual transceiver components through more efficient algorithms (e.g., poly-phase filters) and implementations (e.g., vector-optimized math kernels (Rondeau et al., 2013a)). The runtime environment, i.e., the central component handling the data flow and managing parallel signal processing, did not see similar improvements. This is unfortunate, given the fact that it affects performance critical factors like CPU cache efficiency and overhead from thread synchronization.

The challenge of GPP-based SDRs is to provide a generic runtime environment that offers reasonable performance for any type of transceiver. The requirements are high throughput, low latency, and high stability to avoid occasional data loss. GNU Radio's current approach is to start each transceiver component (like filters or synchronizers) in a separate thread and leave scheduling to the operating system. While the success of GNU Radio shows the practicability of the approach, it also has inherent limitations:

- We cannot control the order in which blocks are scheduled and, therefore, cannot exploit cache coherency.
- We cannot control when and for how long threads are scheduled. A thread might, therefore, be interrupted while holding locks on data, which can lead to suboptimal process sequences.
- We cannot assign multiple blocks to one thread and, therefore, have to use synchronization primitives (like mutexes and semaphores) for all shared data structures.

Proceedings of the 9^{th} GNU Radio Conference, Copyright 2019 by the author(s).

With these limitations, the question arises whether (1) GNU Radio can handle a large number of threads efficiently, i.e., if it scales well with a large number of blocks, and (2) how much performance we lose compared to a more advanced or application-specific scheduler. This paper sheds some light on these questions by providing a performance analysis of the current implementation and a comparison to a manually optimized transceiver. We think that these results help to understand the performance of the GNU Radio runtime environment, thus acting as the base for a discussion about its future development.

2. Related Work

Joseph Mitola's idea of a software defined radio (Mitola, 1995) was realized on GPPs with Vanu Bose's SPECtRA implementation (Bose, 1999), created in the context of the SpectrumWare project at MIT. Bose was motivated by the observation that real-time audio no longer required dedicated hardware but became feasible on normal desktop operating systems. Considering Moore's Law, he assumed that a similar development will take place for radios as well. Yet, if we compare his implementation that was able to sustain a base band sample rate of 33 MHz on a Intel Pentium II with 400 MHz with the performance of current processors and state-of-the-art GPP frameworks, we do not see an equivalent performance increase. This is only partly surprising, since leveraging the potential of today's multi-core CPUs is still a great challenge.

More recent SDR frameworks like Pothosware[1] and Equinox[2] approach this problem with more flexible schedulers that decouple threads from blocks and allow a dynamic assignment. Furthermore, they refrain from using ring buffers[3] but rely on message passing only. While this architecture minimizes scheduling and synchronization overhead, a thorough performance comparison between these frameworks and GNU Radio is still missing.

The potential of a more advanced scheduler can be estimated from manually optimized, application-specific implementations like srsLTE (Gomez-Miguelez et al., 2016) for 4G cellular communications or Microsoft's software radio Sora (Tan et al., 2011), which was released with a GPP-based implementation of IEEE 802.11a/b/g. The latter supports 22 MHz channels and allows the transceiver to respond to an ACK in time. This is impressive, particularly considering that this was running on an Intel Core 2 Duo with 2.67 GHz. Even with more capable PCs, this is still hardly possible with general purpose SDR frameworks.

Another approach to high-performance GPP implementations are Domain-Specific Languages (DSLs) like Ziria (Stewart et al., 2015). The Ziria compiler creates lookup tables automatically, uses vectorized instruction if possible, and performs static scheduling optimizations. While this is a promising approach, its general applicability has yet to be shown. Until now, it did not find widespread adoption and its performance analysis was conducted with a single example that used only two CPU cores. Furthermore, static scheduling optimizations have limitations, as the load per transceiver component might change significantly with the load of the channel, especially for frame-based communications (Bloessl, 2018).

To overcome current constraints of GPP-based SDRs, more and more researchers switch to Field-Programmable Gate Array (FPGA)-based platforms (Khattab et al., 2008) or heterogeneous architectures that combine GPPs with FPGAs (Braun et al., 2016) or Graphics Processing Units (GPUs) (Hitefield & Clancy, 2016; Plishker et al., 2011). While these platforms are more capable, they are still less accessible and have slower development cycles (Sklivanitis et al., 2016). Overall, the systematic performance evaluation of GPP-based SDR platforms is largely unexplored and (reproducible) benchmarks are missing.

3. GNU Radio Run Time Environment

In this paper, we take a first step towards understanding the performance and scaling behavior of a state-of-the-art GPP SDR framework. To this end, we benchmark the scheduler of GNU Radio, a popular, open source, real-time signal processing framework (Rondeau, 2015). With GNU Radio, a transceiver is implemented through a *flowgraph*, which defines the data flow between signal processing *blocks*. These blocks usually correspond to a logical step in the

[1] http://www.pothosware.com/
[2] https://gitlab.com/equinox-sdr/equinox
[3] https://www.gnuradio.org/blog/2017-01-05-buffers/

transceiver like a filter, a synchronizer, or a demodulator. GNU Radio supports two types of data flows between blocks: a stream-based interface, implemented with ring buffers, and a message passing interface. Using ring buffers, a block processes data in its `block::general_work` function, which reads data from input buffers, processes it, and writes the result to the output buffers. Message passing, in turn, is implemented through asynchronous callbacks.

The main advantage of GNU Radio over signal processing libraries implemented for MATLAB or Python is its ability to process a stream of data live while the system is running. This is achieved through highly parallelized data processing, which exploits modern multi-core CPUs by starting each block in a separate thread. Strictly speaking, GNU Radio does not include a real scheduler that manages the threads but relies on the operating system scheduler. The only parameter that is actively controlled is the amount of data that is processed in one step. Unfortunately, there is no obvious optimal setting for this parameter. Generally, data can be processed more efficiently in larger chunks. However, larger chunks also imply that subsequent blocks have to wait longer before they can continue to work with the output. GNU Radio tries do balance this trade-off by filling the output buffer to about 50 %.

4. Run Time Benchmarks

In a first set of experiments, we benchmark the throughput of the GNU Radio scheduler by measuring the run time of a flowgraph when processing a given workload.[4] This is a simple and accurate measure, since it does not require us to modify GNU Radio and introduce measurement probes, which could impact run time behavior and, therefore, the results. Our system uses Ubuntu 19.04 and runs GNU Radio v3.8-rc2, which was compiled with GCC 8.3 in release mode. To conduct stable and reproducible measurements, we use a setup that minimizes interference from the hardware, the operating system, and other processes running on the system. To this end, we create a dedicated *CPU set* that is used exclusively for GNU Radio. The measurements were conducted on a laptop with an Intel i7-8565U processor with eight CPUs (four cores with hyper threads). The CPU set for our performance measurements comprised four CPUs (two cores with their hyper threads). If possible, we also migrate kernel threads and route interrupts to the system's CPU set. We, furthermore, set the CPU governor to *performance* to minimize CPU frequency scaling.

[4]The code and the evaluation scripts for this and the following experiments are available at https://github.com/bastibl/gr-sched.

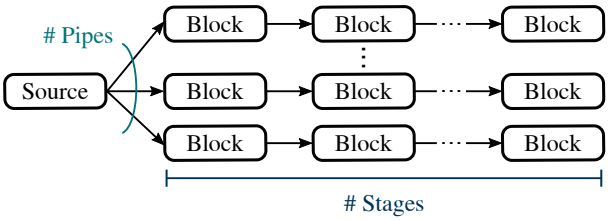

Figure 1. Flowgraph topology, used it our measurements. It allows us to scale the number of parallel streams (*pipes*) and the length of the streams (*stages*).

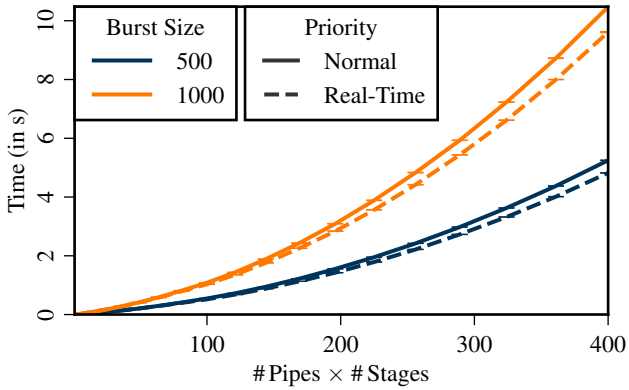

Figure 2. Wall clock time required to process a burst of messages with GNU Radio's message passing interface. Pipes and stages are scaled jointly, i.e., 100 corresponds to 10 pipes and 10 stages.

The flowgraphs in our tests are created programmatically with a C++ application to avoid any possible impact from Python bindings. Our goal is to investigate the scaling behavior when working with a large number of blocks. We, therefore, consider a scenario as shown in Figure 1, where we can configure the number of *pipes* (parallel streams) and *stages* (blocks per stream).

Message Passing Performance

We start by investigating the message passing interface. Here, we create messages in advance and measure the time it takes to forward them through the flowgraph.

Since we want to focus on the performance of the scheduler, we use a custom block, which just forwards messages without any further processing. This custom message forwarder was used for all blocks in Figure 1.

The results are depicted in Figure 2. In this experiment, we scale the number of stages and pipes jointly, i.e., an x-axis value of 100 corresponds to 10 stages and 10 pipes. The y-axis shows the wall clock time,[5] required to run the flowgraph's `top_block::run()` function, i.e., we measure system throughput rather than delay or jitter of individual messages. The burst size indicates the number of 500 Byte messages that we enqueued in the source block before starting the flowgraph. Since all messages are preallocated and since the message passing interface uses pointers to reference messages, the size of the message has limited impact on the results. To shut down the flowgraph, we also enqueue a shutdown message that is processed by GNU Radio's runtime environment. Each data point is based on 50 runs with 10 repetitions per run, i.e., 500 individual measurements. The error bars in this and the following figures indicate the confidence intervals of the mean for a confidence level of 95 %.

As we can see from the plot, the performance scales worse than linearly with the number of blocks in the flowgraph. There is, however, is no severe performance degradation as neither the number of parallel pipes nor the number of stages per pipe cause a problem, even if the number of threads largely exceeds the number of CPU cores (in this case, up to 400 threads on 4 CPU cores). In particular, we were not able to reproduce the poor scaling behavior, presented at SDR Developer Room at FOSDEM 2019,[6] which we believe were conducted with an earlier GNU Radio release, containing a bug that caused blocks to busy-wait for messages. The performance difference between real-time and normal priority is, furthermore, only marginal.

Data-Stream Performance

Apart from message passing, we also benchmarked the data-stream interface with a similar flowgraph topology. Here, the *Source* from Figure 1 was a *Null Source*, followed by a *Head* block, which streamed 100×10^6 floating point numbers (4 Byte each) into the flowgraph. To focus on the runtime and not the performance of signal processing blocks, we used *Copy* blocks in the pipes, which just copy data from their input to their output buffer. Each pipe was, furthermore, terminated by a *Null Sink*. Like in the previous experiment, we scaled the number of pipes and stages jointly.

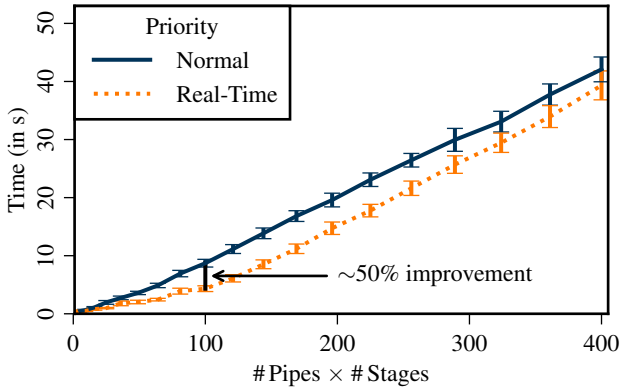

Figure 3. Wall clock time required to process 100×10^6 samples with GNU Radio streaming interface. Pipes and stages are scaled jointly, i.e., 100 corresponds to 10 pipes and 10 stages.

The results for real-time and normal priority are shown in Figure 3. In this case, we noticed a considerable difference between priority settings. A setup with 100 blocks, for example, runs approximately 50 % faster with real-time priority than with normal priority. This was an unexpected result, since we assumed that real-time priority asserts that GNU Radio threads are scheduled before any other threads of the operating system. However, given the fact that we use a CPU set exclusively for GNU Radio, there are no priority issues.

5. Linux Task Scheduling

The deviating results can be explained by the fact that Linux uses different schedulers for different thread types. Figure 4 shows the scheduler hierarchy of a modern Linux system. It comprises three classes, which are (in decreasing priority): the *Deadline* scheduler, the *Real-Time* scheduler, and the normal scheduler. Schedulers with a lower priority are generally only run when higher priority schedulers have no active threads. Yet, the normal Linux scheduler gets a guaranteed run time (typically 50 ms per second) to avoid that defective

[5] With wall clock time, we refer to the time that it takes to execute the function and contrast, for example, the (aggregated) time that threads were scheduled by the operating system.

[6] https://fosdem.org/2019/schedule/event/sdr_equinox/

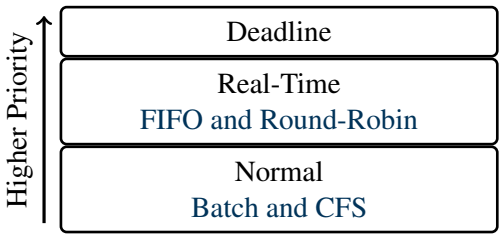

Figure 4. Linux process scheduling hierarchy.

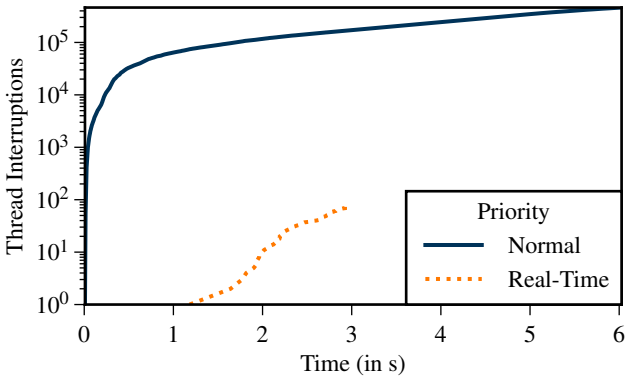

Figure 5. Cumulative number of thread interruptions over time of a flowgraph with ten pipes and ten stages.

real-time threads lock-up the system. The *Deadline* scheduler is used for sporadic real-time tasks with hard deadlines and deterministic run time, which is of limited interest for SDR.

When we configure real-time priority with GNU Radio, the threads are handled by the real-time scheduler. Real-time threads have a priority between 1 (high) and 99 (low), similar to the *nice* value. However, with the real-time scheduler, threads with a higher priority do not get more CPU time but are always processed before threads with a lower priority. The real-time scheduler supports two scheduling algorithms: Round Robin and FIFO. With FIFO, threads are processed in a first-in-first-out manner until they are done or a task with a higher priority gets active. The round robin scheduler, in turn, schedules task with similar priorities for a fixed time slice (by default 100 ms). While GNU Radio supports both algorithms, it uses the round robin scheduler by default.

The normal scheduler supports two algorithms: the Completely Fair Scheduler (CFS) and a batch scheduler. The latter was added to handle CPU-intensive background tasks efficiently. Threads from this scheduler are only run if no other threads are active on the system. The main advantage of a dedicated batch scheduler is (1) it does not interfere with interactive tasks and (2) it offers low overhead, since threads can be scheduled longer, which minimizes context switches. The CFS is the default scheduler, which is used for the vast majority of tasks. It is by far the most complex algorithm that tries to balance between interactivity, efficiency, and fairness. Each active thread gets CPU time depending on past CPU time, the number of threads in the run queue, and its nice value.

Considering GNU Radio, the performance difference between normal and real-time priority can be explained by the CPU time that each thread is assigned, once it is scheduled. The round robin scheduler always schedules a block for 100 ms, which might be enough to process all samples. The CFS, in turn, assigns CPU time depending on the number of active threads, i.e., the number of GNU Radio blocks that have samples in their input queues and space in their output queues. The CPU time might, therefore, be much smaller and a block might not be able to process all samples before it is interrupted. Since the block is not finished, it also did not advance the pointers of its ring buffers, so that adjacent blocks might only be able to process a small amount of samples or are not able to continue at all. Overall, this might result in less optimal processing sequences.

6. Profiling the Scheduler

To better understand the impact of the scheduler, we use the Linux *perf* tool to profile the Linux scheduler while the flowgraph is running. While GNU Radio exports per block performance statistics through *Performance Counters* (Rondeau et al., 2013b), they focus on performance bottlenecks and efficient block implementations. For profiling the scheduler, they are of limited use. In the first experiment, we focus on the impact of thread interruptions by the scheduler. To this end, we record the Linux tracepoint *sched_switch*, which is called when the scheduler switches between tasks. Among other things, the tracepoint logs a timestamp, the thread that is scheduled out, and the previous state of the thread, i.e., if the thread was interrupted or

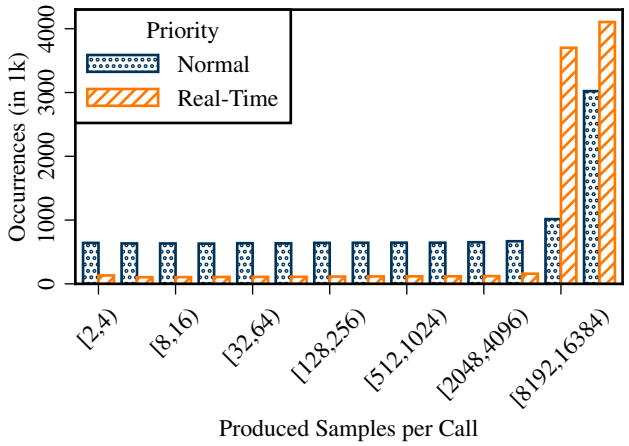

Figure 6. Distribution of the number of samples that were produced in one call to `general_work` by a flowgraph with ten pipes and ten stages.

if it entered a sleep state. The first case means that the block execution was interrupted while processing samples, the latter case means that the block processed all samples and waits for upstream or downstream blocks to provide more samples or free buffer space, respectively. We profile a flowgraph with ten pipes and ten stages that processes 100×10^6 floating-point numbers, similar to the previous experiments. Note that, while a tracepoint is a low-overhead method to profile Linux internals, adding a probe inevitably changes the system and with it the result. We, therefore, took care to minimize the impact of the profiler and made sure that the perf kernel buffer is large enough to fit all events from one run, i.e., perf does not have to copy data to disk while the flowgraph is running.

The cumulative number of thread interruptions over time is plotted in Figure 5. We can see a huge difference between normal and real-time priority. (Note the logarithmic y-axis.) As expected, the CPU time provided by the real-time scheduler is nearly always enough to process all samples. The CFS scheduler, in turn, often interrupts blocks while processing samples.

Since we assumed that this can lead to suboptimal scheduling sequences, we also compare the distribution of the number of items that are processed by `block::general_work`. Here, we use *uprobes*, i.e., dynamic user space probes that can be attached to shared library functions. While uprobes can also be instrumented through perf, we decided to use BPF Compiler Collection (BCC)[7], which provides an interface to Linux extended Berkeley Packet Filter (eBPF), a more recent framework to collect performance data. Its main advantage over perf is that it allows to process and aggregate data while it is recorded, i.e., we do not have to log raw event data. We hook a probe to the return of `general_work` and log the number of items that were produced per call.

The distribution over a period of 20 s is depicted in Figure 6. It shows that the default CFS scheduler produces much more often a small number of samples, which can cause considerable overhead, especially since every call of to `work()` requires accessing shared data structures. Overall, these experiments show that the scheduler can have a significant impact on the performance of a flowgraph. In particular, short time slices cause blocks to be scheduled out while in work, which results in inefficient workloads for GNU Radio blocks.

7. Optimizing Flowgraphs

We have already seen that the operating system scheduler can have a major impact on the performance of a flowgraph. In addition to that, there are further GNU Radio optimizations with a potential impact on the runtime performance. The most interesting ones in this context are the buffer size and the maximum number of samples that are produced in one call to `block::general_work`. To optimize locality and benefit from CPU caches, an ideal scheduler would run a block to produce samples corresponding to the cache size, before running the downstream block on the same CPU core. By adjusting the buffer size and the maximum number of samples that are produced, we can limit the data that is produced in one call to `work` to the cache size. While this gives downstream blocks *the chance* to run, we cannot guarantee that these blocks are scheduled back to back on the same CPU core.

Even though a new general purpose scheduler is out of the scope of this paper, we wanted to get an idea of the potential that manual optimizations and an advanced scheduler could provide. To this end, we consider a simple flowgraph topology that is easy to optimize manually. For a more complex general flowgraph, this would not easily be possible. As shown in Figure 7, we create four independent branches that forward samples

[7] https://github.com/iovisor/bcc

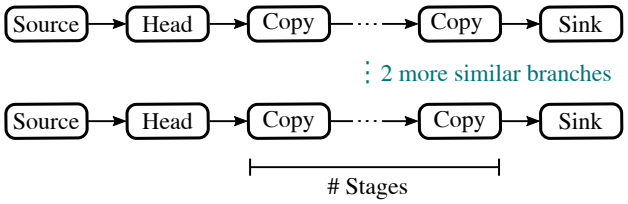

Figure 7. Simple flowgraph topology, used to benachmark throughput of different optimizations.

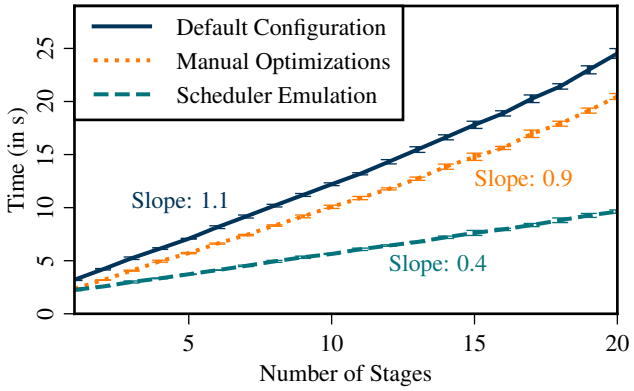

Figure 8. Flowgraph run time depending on the number of blocks for different optimization levels. The slope is calculated with a linear regression.

through a given number of *stages*. In each branch, the samples are generated from a Null Source. The number of samples that traverse a branch is limited by a Head block, which forwards 1×10^9 samples. Given the simple flowgraph layout, it is straightforward to select better parameters. We consider three configurations:

- **Default**: The default configuration of GNU Radio, i.e., no CPU affinity, a buffer size of 64 kByte, and a maximum of 100 000 items that are produced in one call to `work`.
- **Manual Optimization**: We pin the blocks of a branch to the same CPU core, adjust the buffer to twice the size of the L1 cache (since GNU Radio tries to fill 50 % of the buffer), and configure a maximum number of output items corresponding to the size of the L1 cache.
- **Scheduler Emulation**: We emulate a more advanced scheduler by replacing the copy operations with a single block that performs similar copy operations internally. It copies chunks of samples sequentially through buffers, with the chunk size corresponding to the size of the L1 cache.

Note that also the third configuration is not strictly optimal, since the source and head blocks are still scheduled independently. Copying is, therefore, interrupted randomly in favor of source and head blocks, which might thrash the cache. The sink block is no problem here, since it does not perform any memory operations.

Again, we measure the run time of the flowgraph, by timing the `top_block::run()` function to exclude flowgraph setup. Like in the previous experiments, we allocate four CPU cores exclusively to GNU Radio. Furthermore, we use real-time priority for all configurations and disable compiler optimizations for our custom copy block to make sure that no copy operations are optimized out.

The results of the experiment are shown in Figure 8, where we plot the run time against the number of stages. Two things are interesting in this plot: First, we can see that all configurations scale linearly, even for up to 20 stages per CPU core. Second, while manual optimization provides a slight advantage over the default configuration, a much bigger benefit is promised by a more advanced scheduler. The improvement is the combined effect of (1) less synchronization overhead, since there is only one thread for all copy operations, and (2) better cache utilization, since threads are not scheduled in random order.

8. Conclusion

In this paper, we have benchmarked the throughput of the GNU Radio runtime environment, in particular, its scaling behavior for a large number of blocks. We have shown that both message passing and buffer-based flowgraphs scale well with the number of blocks. This is a positive results, given the concerns that the performance of the current implementation could diminish when the number of blocks exceeds the number of CPU cores. We have, furthermore, seen that the operating system scheduler can have a major impact and switching from the default to the real-time scheduler can reduce the runtime by up to 50 %. Yet, we have also shown the limitations of the current implementation by emulating a more advanced scheduler for a

simple flowgraph topology. The fact that this configuration scales linearly with a much lower slope gives an impression of what could be possible.

Acknowledgment

This work has been supported by the DFG within SFB 1053 MAKI. It has been performed in the context of the LOEWE center emergenCITY.

References

Bloessl, Bastian. *A Physical Layer Experimentation Framework for Automotive WLAN*. PhD Thesis (Dissertation), Paderborn University, June 2018.

Bose, Vanu G. *Design and Implementation of Software Radios using a General Purpose Processor*. PhD Thesis, Massachusetts Institute of Technology, April 1999.

Braun, Martin, Pendlum, Jonathan, and Ettus, Matt. RFNoC: RF Network-on-Chip. In *6th GNU Radio Conference*, Boulder, CO, September 2016. GNU Radio Foundation.

Gomez-Miguelez, Ismael, Garcia-Saavedra, Andres, Sutton, Paul D., Serrano, Pablo, Cano, Cristina, and Leith, Doug J. srsLTE: An Open-Source Platform for LTE Evolution and Experimentation. In *10th ACM International Workshop on Wireless Network Testbeds, Experimental evaluation and Characterization (WiNTECH 2016)*, New York City, NY, Oct 2016. ACM.

Hitefield, Seth and Clancy, T. Charles. Flowgraph Acceleration with GPUs: Analyzing the Benefits of Custom Buffers in GNU Radio. In *6th GNU Radio Conference*, Boulder, CO, September 2016. GNU Radio Foundation.

Khattab, Ahmed, Camp, Joseph, Hunter, Chris, Murphy, Patrick, Sabharwal, Ashutosh, and Knightly, Edward W. WARP: A Flexible Platform for Clean-Slate Wireless Medium Access Protocol Design. *ACM SIGMOBILE Mobile Computing and Communications Review*, 12(1), Jan 2008.

Mitola, Joseph. The Software Radio Architecture. *IEEE Communications Magazine*, 33(5), May 1995.

Plishker, William, Zaki, George F., Bhattacharyya, Shuvra S., Clancy, Charles, and Kuykendall, John. Applying Graphics Processor Acceleration in a Software Defined Radio Prototyping Environment. In *22nd IEEE International Symposium on Rapid System Prototyping (RSP)*, Karlsruhe, Germany, May 2011. IEEE.

Rondeau, Thomas W. On the GNU Radio Ecosystem. In Holland, Oliver, Bogucka, Hanna, and Medeisis, Arturas (eds.), *Opportunistic Spectrum Sharing and White Space Access: The Practical Reality*. Wiley, May 2015.

Rondeau, Thomas W., McCarthy, Nicholas, and O'Shea, Timothy. SIMD Programming in GNU Radio: Maintainable und User-Friendly Algorithm Optimization with VOLK. In *SDR-WInnComm 2013*, Washington, DC, January 2013a. Wireless Innovation Forum.

Rondeau, Thomas W., O'Shea, Timothy, and Goergen, Nathan. Inspecting GNU Radio Applications with ControlPort and Performance Counters. In *ACM SIGCOMM 2013, 2nd ACM SIGCOMM Workshop of Software Radio Implementation Forum (SRIF 2013)*, Hong Kong, China, Aug 2013b. ACM.

Sklivanitis, George, Gannon, Adam, Batalama, Stella N., and Pados, Dimitris A. Addressing Next-Generation Wireless Challenges with Commercial Software-Defined Radio Platforms. *IEEE Communications Magazine*, 54(1), Jan 2016.

Stewart, Gordon, Gowda, Mahanth, Mainland, Geoffrey, Radunovic, Bozidar, Vytiniotis, Dimitrios, and Agullo, Cristina Luengo. Ziria: A DSL for Wireless Systems Programming. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems*, Istanbul, Turkey, 3 2015. ACM.

Tan, Kun, Liu, He, Zhang, Jiansong, Zhang, Yongguang, Fang, Ji, and Voelker, Geoffrey M. Sora: High Performance Software Radio Using General Purpose Multi-core Processors. *Communications of the ACM*, 54(1), January 2011.