A Flexible and Portable Real-time DVB-S2 Transceiver using Multicore and SIMD CPUs

Adrien Cassagne, Mathieu Leonardon, Romain Tajan, Camille Leroux, Christophe Jégo, Olivier Aumage, Denis Barthou

A Flexible and Portable Real-time DVB-S2 Transceiver using Multicore and SIMD CPUs

Adrien Cassagne*†, Mathieu Léonardon‡, Romain Tajan*, Camille Leroux*, Christophe Jégo*,
Olivier Aumage† and Denis Barthou†
*University of Bordeaux, Bordeaux INP, IMS Lab, UMR CNRS 5218, France
†University of Bordeaux, Bordeaux INP, Inria / LaBRI, France
‡IMT Atlantique, Lab-STICC, UMR CNRS 6285, F-29238 Brest, France

***Abstract*—Software implementation of digital communication systems is more and more used in different contexts. In the case of satellite communication standards, they are an appealing alternative in ground stations. The challenge is to push the performance of these digital communication systems to meet the real time constraints. In this paper, we propose an open source digital communication transceiver that enables to exploit the parallelism of general purpose processors (multicore, SIMD). It is also flexible, supporting several modulation and coding schemes. Finally, it is portable, being able to adapt to the level of parallelism of different CPU architectures (x86 and ARM).**

***Index Terms*—Real-time system, SDR, Multicore CPU, SIMD, DVB-S2 standard, Radio transceiver**

I. Introduction

Software implementation of digital communication standards is a high-stakes research path for communication network players. It is promising from many points of view: hardware mutualization, energy savings, maintainability and scaling. It is also an attractive choice when the production volume is low and does not enable to compensate for non-recursive engineering costs. In the case of the DVB-S2 standard, on which this article focuses, ground stations are good candidates. However, the existing solutions for implementing the DVB-S2 transceiver in software are not sufficient. They are either flexibility oriented, as for example by using the GNU radio software suite [1], but at the expense of performance, not suitable with real time processing. On the other hand, highly optimized solutions exist, but they sacrifice flexibility [2]. Moreover, the code is not available, which limits its reusability and reproducibility. The main objective of this paper is to provide a clear, structured, open and efficient code base, for the implementation of all the elementary blocks of the DVB-S2 system, from transmission to reception through the signal acquisition and synchronization steps. The rest of the paper is organized as follows. Sec. II presents the communication system and its different parts. Sec. III details the real time constraints and the optimizations that have been implemented to achieve them, including multi-threading and SIMD parallelization. Finally, Sec. IV presents the related works.

II. DVB-S2 Transceiver

The second generation of Digital Video Broadcasting standard for Satellite (DVB-S2) [3] is a flexible standard designed for broadcast applications. DVB-S2 is typically used for the digital television (HDTV with H.264 source coding). The full DVB-S2 transmitter and receiver are implemented in a SDR-compliant system. Two Universal Software Radio Peripherals (USRPs) N320 have been used for the analog signal transmission and reception while all the digital processing of the system have been implemented on CPU.

A. Transmitter Software Implementation

Fig. 1 shows the DVB-S2 transmitter decomposition in tasks. The filled tasks are intrinsically sequential. The initial information bits are read from a binary file (t_1^{Tx}). Then, the DVB-S2 coding scheme rests upon the serial concatenation of a BCH code (t_3^{Tx}) and an LDPC code (t_4^{Tx}). The selected modulation (t_6^{Tx}) is a Phase-Shift Keying (PSK). The scrambler tasks (t_2^{Tx} and t_8^{Tx}) apply predefined repeated sequences of *xor* to the frame in order to avoid overly long sequences of the same bit or symbol in the frames sent by the radio (t_{10}^{Tx}). Depending on the DVB-S2 MODulation and CODing scheme (MODCOD), the frame can be interleaved (t_5^{Tx}) after the encoding process. If there is no interleaver, the frame is just copied. After the modulation, PayLoad Header (PLH) and pilots are inserted (t_7^{Tx}). These extra data are used by the synchronization tasks at the receiver side. Before the radio transmission (t_{10}^{Tx}), the signal bandwidth is rescaled by a shaping filter (t_9^{Tx}).

TABLE I
Selected DVB-S2 configurations (MODCOD).

Configuration	Modulation	K	K_{LDPC}	Rate R
MODCOD 1	QPSK	9552	9720	3/5
MODCOD 2	QPSK	14232	14400	8/9
MODCOD 3	8-PSK	14232	14400	8/9

The DVB-S2 standard contains 32 different configurations or *MODCODs*. This work focuses on 3 typical MODCODs given in Tab. I. Depending on the MODCOD, the PSK modulation and the LDPC code rate R vary. In MODCOD 1 and 2 there is no interleaver, MODCOD 3 includes a column/row interleaver. K is the number of information bits and the input size of the BCH encoder. K_{LDPC} is the output size of the BCH encoder and the input size of the LDPC encoder. The LDPC codeword size is constant for the 3 MODCODs ($N_{\text{LDPC}} = 16200$).

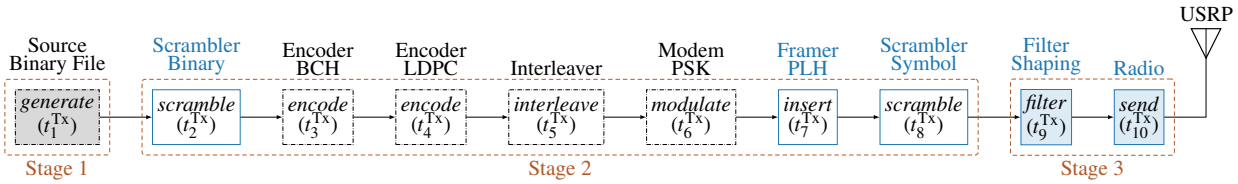

Fig. 1. DVB-S2 transmitter software implementation.

B. Receiver Software Implementation

Fig. 2 and Fig. 3 describe the tasks decomposition of the DVB-S2 receiver software implementation with five distinct phases. The first one is called the *waiting phase* (see Fig. 2). It consists in waiting until a transmitter starts to transmit. The *Synchronizer Frame* task (t_8^{Rx}) possesses a frame detection criterion. When a signal is detected, the *acquisition (acq.) phase 1* (see Fig. 2) is executed during 150 frames. After that, the *acq. phase 2* (see Fig. 2) is also executed during 150 frames. After the *acq. phase 1 and 2*, tasks have to be re-bound for the *acq. phase 3* (see Fig. 3). This last acq. phase is applied over 200 frames. After the 500 frames of these acq. phases, the final *transmission phase* is established (see Fig. 3).

In a real life transmission systems, radios internal clocks can drift slightly. A specific processing has to be added in order to be resilient. This is achieved by the *Synchronizer Timing* tasks (t_5^{Rx} and t_6^{Rx}). Similarly, the radio transmitter carrier frequency does not perfectly match the receiver carrier frequency. So, the *Synchronizer Frequency* tasks (t_3^{Rx}, t_{10}^{Rx} and t_{11}^{Rx}) aims at estimating and compensating the carrier frequency offset and the phase offset. Finally, LDPC FEC is a block coding scheme that requires to know precisely the first and last bits of a codeword. The *Synchronizer Frame* task (t_8^{Rx}) uses the PLH and pilot symbols inserted by the transmitter (t_7^{Tx}) to recover the first and last symbols. One can notice that the *Synchronizer Timing* module is composed by two separated tasks (*synchronize* or t_5^{Rx} and *extract* or t_6^{Rx}). This behavior is different from the other *Synchronizer* modules. The *synchronize* task (t_5^{Rx} or $t_{3,4,5}^{Rx}$) has two outputs: one for the regular data and another one for a mask. The regular data and the mask are then used by the *extract* task (t_6^{Rx}) to screen which data that is selected or not for the next task. This specific implementation has been retained for two reasons. Firstly, the *Synchronizer Timing* tasks (t_5^{Rx} and t_6^{Rx}) have a high latency compared to the others tasks, thus splitting the treatment in two tasks is a way to increase the throughput of the pipeline (this is further discussed below). Secondly, the *extract* task (t_6^{Rx}) introduces a new possible behavior. In some cases the task does not have enough samples to produce a frame. In such cases, the *extract* task raises an exception. When this exception is caught, the chain restarts from the first task (t_1^{Rx}). This implies to manage a buffer of samples in the *extract* task (t_6^{Rx}). If the buffer contains more than one frame then the next task (t_7^{Rx}) is executed, otherwise the chain is restarted.

Fig. 4 shows the FER decoding performance results of the 3 selected MODCODs. The shapes represent the channel conditions: squares stand for a standard simulated AWGN channel, triangles are also a simulated AWGN channel in which frequency shift, phase shift and symbol delay have been taken into account, circles are the real conditions measured performances with the two USRPs. One can notice a 0.2 dB inaccuracy in the noise estimated by the t_{13}^{Rx} task. It is symbolized by the extra horizontal bars over the circles. For each MODCOD, the LDPC decoder is based on the belief propagation algorithm with horizontal layered scheduling (10 ite.) and with the min-sum node update rules. Each DVB-S2 configuration has a well-separated SNR predilection zone.

C. Open Source Integration with AFF3CT Toolbox

The proposed software implementation of the DVB-S2 digital transceiver is open source and available online[1]. It is described with the help of the AFF3CT toolbox [4]. AFF3CT is a library dedicated to the digital communication systems and more specifically to the channel decoding algorithms. At the time of the writing, the project focuses more on functional simulations/evaluations than on real-time digital communication systems. In this paper, we extend AFF3CT to the SDR use case while keeping the interoperability, reproducibility and maintainability philosophy initiated in the toolbox.

Some components are directly used from the AFF3CT library (black dashed-dotted tasks in Fig. 1 and Fig. 3). Thus, the software implementations are optimized for efficiency and the features have been checked. For instance, knowing that the LDPC decoding is one of the most compute intensive task, an existing high performance SIMD implementation is used. This implementation decodes multiple frames in parallel to reach very high throughputs. The *Decoder LDPC* task (t_{16}^{Rx}) is the only one in the receiver that takes advantage of the multiple frames to fill the SIMD registers (inter-frame SIMD strategy). The other tasks simply process multiple frames sequentially. This implementation choice negatively affects the latency of these tasks by a factor equal to the number of frames. But, in the targeted video streaming app., the latency is not important.

New tasks have been implemented specifically for this project (blue plain boxes in Fig. 1, 2 and 3). They have been described according to the AFF3CT interfaces. Thus, allowing to take advantage of instrumentation for fine-grained throughput and latency measurements, as well as tools to facilitate debugging. The proposed implementations are mainly focusing on two missing aspects in AFF3CT: 1) signal synchronizations and filters, 2) real-time communications. Being open source,

[1]DVB-S2 digital transceiver repository: https://github.com/aff3ct/dvbs2.

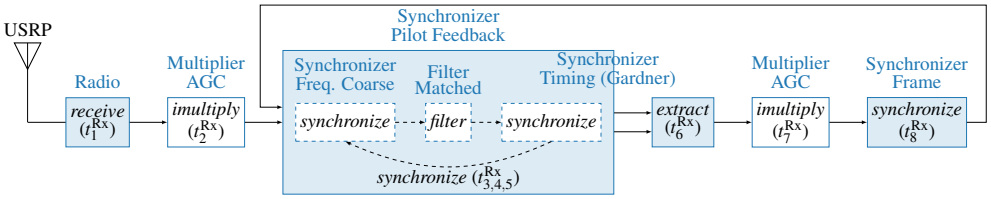

Fig. 2. DVB-S2 software receiver: waiting phase and acquisition phase 1 & 2.

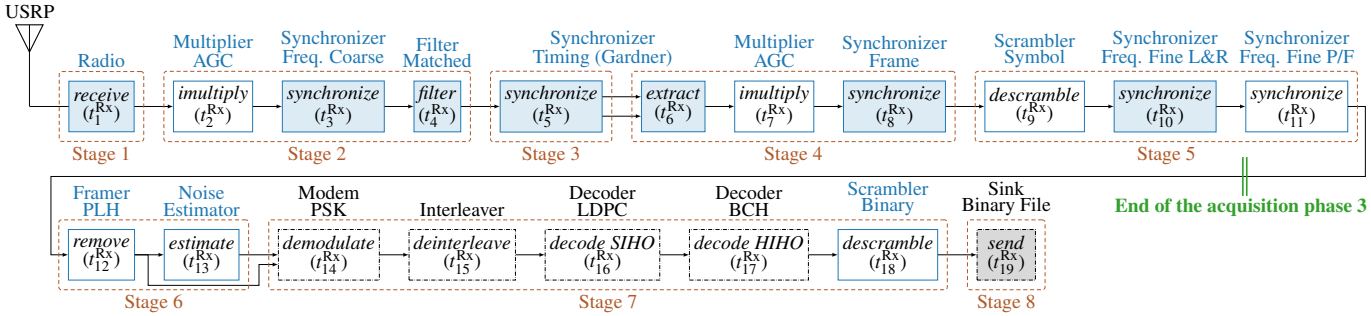

Fig. 3. DVB-S2 software receiver: acquisition phase 3 & transmission phase.

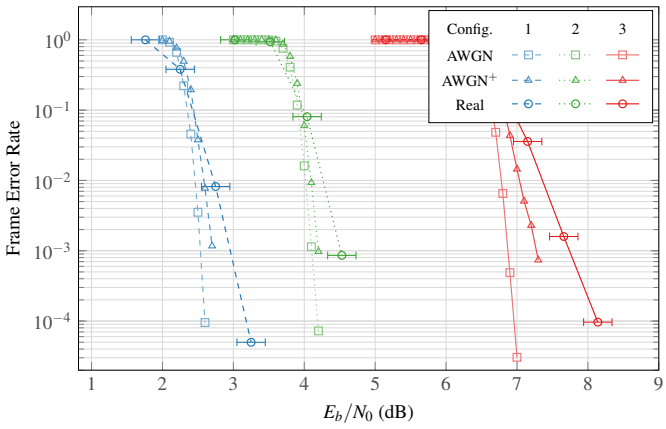

Fig. 4. DVB-S2 FER decoding performance.

all of these software implementations, that are optimized for real-time processing, are reusable for the community.

III. Real Time Implementation Features

A. Reliable Sample Collection and Delivery

The processing rate of the whole system is determined by the rate (in MS/s) at which USRP modules are set to run. The transmitter part has to provide samples at a sufficient rate, whereas the receiver part must consume them. In our study, the real time constraint was imposed by our industrial partner whose goal was that the complete chain reach several tens of MB/s. Furthermore, there should be a very limited jitter in the samples delivering and gathering rates at both sides. Indeed, the inner buffers of the USRP modules are limited. Precautionary measures are necessary to avoid temporary rate drops, that would cause underflows at the transmitter side, or overflows at the receiver side. Firstly, the corresponding threads are pinned to a dedicated core (in t_{10}^{Tx} and in t_1^{Rx}). Then, preemption of these threads has to be avoided. To this purpose, we chose to isolate the CPU core to which these threads are pinned, using the `isolcpus` Linux kernel option.

Provided with sufficient buffers on the host side, these threads efficiently handle the data exchange between the host and the USRP modules. Accessing real time processing is now just a matter of getting sufficient throughput for other tasks.

The transmitter is not the most resources consuming part that is why delivering high data rate is not as challenging as the receiver counterpart. The DVB-S2 transmitter software implementation has been split into 3 pipeline stages. Stages 1 and 3 are sequential, stage 2 is parallel. An Intel® Core™ i7 CPU with 4 cores (SMT was switched on) has been selected (Haswell architecture). One entire core has been reserved for the radio thread (stage 3), one hardware thread has been assigned to the *source* (stage 1) and the five remaining hardware threads have been dedicated to the stage 2. The pipeline implementation is a copy-less and lock-free producer/consumer using a passive waiting strategy.

B. SIMD Implementations of Synchronization and Filter Steps

The synchronization and filter tasks extensively rely on vectorized implementations. SIMD instructions are used to speedup the processing inside a frame (intra-frame SIMD strategy). The synchronization tasks are working on complex numbers. Actually, we have chosen to represent these numbers as an Array of Structures (AoS) in memory. To guarantee the portability and the flexibility of the code, the MIPP SIMD library [5] is used. Additional reordering operations (`mipp::deinterleave` and `mipp::interleave`) were necessary to manage the AoS representation. The filter task (t_4^{Rx}) mainly perform linear algebra operations. Thus, the proposed implementation extensively rely on efficient *Fused Multiply-Add* (FMA) instructions (`mipp::fmadd`).

TABLE II
RECEIVER SEQUENTIAL TASK CHARACTERISTICS ON THE X86 CPU. SEQUENTIAL TASKS ARE REPRESENTED BY BLUE ROWS. THE SLOWEST SEQ. STAGE IS IN RED WHILE THE SLOWEST OF ALL IS IN ORANGE.

Stages and Tasks	Throughput (Mb/s)	Latency (μs)	Time (%)
Radio - *receive* (t_1^{Rx})	431.83	527.32	0.94
Stage 1	431.83	527.32	0.94
Multiplier AGC - *imultiply* (t_2^{Rx})	367.45	619.71	1.11
Synch. Freq. Coarse - *synchronize* (t_3^{Rx})	841.32	270.66	0.48
Filter Matched - *filter* (t_4^{Rx})	116.41	1956.08	3.49
Stage 2	80.00	2846.45	5.08
Synch. Timing - *synchronize* (t_5^{Rx})	55.42	4108.52	7.34
Stage 3	55.42	4108.52	7.34
Synch. Timing - *extract* (t_6^{Rx})	281.83	807.97	1.44
Multiplier AGC - *imultiply* (t_7^{Rx})	685.51	332.18	0.59
Synch. Frame - *synchronize* (t_8^{Rx})	159.41	1428.51	2.55
Stage 4	88.65	2568.66	4.58
Scrambler Symbol - *descramble* (t_9^{Rx})	1682.89	135.31	0.24
Synch. Freq. Fine L&R - *synchronize* (t_{10}^{Rx})	1246.85	182.63	0.33
Synch. Freq. Fine P/F - *synchronize* (t_{11}^{Rx})	112.56	2022.98	3.61
Stage 5	97.27	2340.92	4.18
Framer PLH - *remove* (t_{12}^{Rx})	1008.60	225.77	0.40
Noise Estimator - *estimate* (t_{13}^{Rx})	550.06	413.98	0.74
Stage 6	355.94	639.75	1.14
Modem PSK - *demodulate* (t_{14}^{Rx})	40.47	5626.34	10.05
Interleaver - *deinterleave* (t_{15}^{Rx})	1347.25	169.02	0.30
Decoder LDPC - *decode SIHO* (t_{16}^{Rx})	164.21	1386.74	2.48
Decoder BCH - *decode HIHO* (t_{17}^{Rx})	6.92	32905.37	58.79
Scrambler Binary - *descramble* (t_{18}^{Rx})	91.11	2499.41	4.47
Stage 7	5.35	42586.88	76.09
Sink Binary File - *send* (t_{19}^{Rx})	1838.31	123.87	0.22
Stage 8	1838.31	123.87	0.22
Total	4.09	55742.37	99.57

C. High Throughput Receiver

This section details the receiver part of the system in which achieving high data rates is clearly challenging. The presented results have been obtained on two high-end NUMA machines. The first one is composed by two Intel® Xeon™ Platinum 8168 CPUs (denoted as x86). The frequency of the CPUs is 2.70 GHz (24 cores, 128 GB of RAM) and the *Turbo Boost* mode has been disabled for the reproducibility of the experiment results. Each core is powered by AVX-512F SIMD ISA. The second architecture is composed by two Cavium ThunderX2® CN9975 v2.1 CPUs (denoted as ARM). The frequency of the CPUs is 2.00 GHz (28 cores, 256 GB of RAM). Each core is powered by NEON SIMD ISA. In both targets the SMT was switched off. In the proposed implementation, the data are represented by 32-bit numbers. Thus, the data parallelism level is 16 for AVX-512F ISA and 4 for NEON ISA.

Tab. II presents the task throughputs and latencies measured from a sequential execution of the MODCOD 2 in the transmission phase and on the x86 target. The tasks have been regrouped per stage in order to introduce the future decomposition when the parallelism is applied. The throughputs have been normalized to the number of information bits (K). This enables the comparison among all the reported throughputs.

The stage 7 takes 76% of the time with especially the *Decoder BCH* task (t_{17}^{Rx}) that takes 59% of the time. t_{17}^{Rx} should not take so much time compared to the other tasks. However, we chose to not spend time in optimizing the BCH decoding process as the stage 7. Indeed, throughput can easily be increased by running the tasks on multiple threads. The second slower stage is the stage 3. This stage is the main hotspot of the implemented receiver. The stage 3 contains only one synchronization task (t_5^{Rx}). In the current implementation, this task cannot be duplicated (or parallelized) because there is an internal data dependency with the previous frame (state-full task). The stage 3 is the real limiting factor of the receiver. For information, considering a CPU with an infinite number of cores, the maximum reachable throughput is 55.42 Mb/s.

We did not try to parallelize the waiting and the acq. phases. We measured that the whole acq. phase (1, 2 and 3) takes about one second (on x86). During the acq. phase, the receiver is not fast enough to process the received samples in real time. To fix this problem, the samples are buffered in the *Radio - receive* task (t_1^{Rx}). Once the acq. phase is done, the transmission phase is parallelized. Thus, the receiver becomes fast enough to absorb the radio buffer and samples in real time. During the transmission phase, the receiver is split into 8 consecutive pipeline stages as presented in Fig. 3. This decomposition has been motivated by the nature of the tasks (sequential or parallel) and by the sequential measured throughput. The number of stages has been minimized in order to limit the pipeline overhead. Consequently, sequential and parallel tasks have been regrouped in common stages. The slowest sequential task (t_5^{Rx}) has been isolated in the dedicated stage 3. The other sequential stages have been formed to always have a higher normalized throughput than the stage 3. The sequential throughput of the stage 7 (5.35 Mb/s) is lower than the throughput of the stage 3 (55.42 Mb/s). This is why we duplicated this stage to run over 28 threads. This looks overkill but the machine was dedicated to the DVB-S2 receiver and the throughput of the *Decoder LDPC* task (t_{16}^{Rx}) varies depending on the SNR. One can notice that an early termination criterion was activated. When the signal quality is very good, the *Decoder LDPC* task runs fast and the threads can spend a lot of time in waiting. In Tab. II, the presented *Decoder LDPC* task throughputs and latencies are optimistic because we are in a SNR error-free zone. All the threads are pinned to a single core with the *hwloc* [6] library. The 28 threads of the stage 7 are pinned in round-robin between the CPU sockets. By this way, the memory bandwidth is maximized thanks to the two NUMA memory banks. During the duplication process, the thread pinning is known and the data are copied into the right memory bank (first touch policy). All the other pipeline stages (1, 2, 3, 4, 5, 6 and 8) are running on a single thread. Because of the synchronizations between the pipeline stages, the threads have been pinned on the same socket. The idea is to minimize the pipeline stage latencies in maximizing the CPU cache performance. It avoids the extra-cost of moving the cache data between the sockets. On the ARM target, the pipeline has been decomposed in 12 sequential stages and 1 parallel stage of 40 threads (stage 7).

The receiver program assigned around 1.3 GB of the global memory when running in sequential while it assigned around

TABLE III
THROUGHPUTS DEPENDING ON THE SELECTED DVB-S2 CONFIGURATION.

	Throughput (Mb/s)					
	Sequential		Parallel		Latency (ms)	
Configuration	x86	ARM	x86	ARM	x86	ARM
MODCOD 1	3.4	1.0	37	19	–	37
MODCOD 2	4.1	1.4	55	28	56	41
MODCOD 3	4.0	1.1	80	42	–	51

30 GB in parallel. The memory usage increases because of the chain duplications in the stage 7. The duplication operation takes about 20 seconds on the x86 and ARM targets. It is made at the very beginning of the program. It is worth mentioning that the amount of memory was not a critical resource.

Tab. III summarizes the obtained throughputs for the 3 MODCODs presented in Tab. I. Each time, sequential and parallel throughput are given. To measure the maximum achievable throughput, the USRP modules have been replaced by pre-registered samples. This is because the pipeline stages are naturally adapting to the slowest one. It means that in a real communication, the throughput of the radio is always configured according to the slowest stage. Indeed, it is necessary for real time communication otherwise the radio task has to indefinitely buffer the samples while the amount of available memory in the machine is finite. The throughput value is the final useful information throughput (K bits) for the user. Between the MODCOD 1 and 2, only the LDPC code rate varies ($R = 3/5$ and $R = 8/9$ resp.). It has a direct impact on the information throughputs. Between the MODCOD 2 and 3, the modulation varies (QPSK and 8-PSK resp.) and the frames have to be deinterleaved (column/row). High order modulation reduces the amount of samples processed by the *Synchronizer Timing* task (t_5^{Rx}). This results in higher throughput in the slowest stage 3 (80 Mb/s for the 8-PSK on the x86 target). In the parallel implementations, the pipeline stage throughputs are adapting to the slowest stage 3. It results in an important speedup. In the sequential implementations, we observe a slowdown. This is mainly due to the *demodulate* task (t_{14}^{Rx}) which takes a higher amount of time. This additional demodulation cost is absorbed by the 28 threads of the stage 7 when running in parallel. For MODCOD 2 in parallel, one can note that the achieved information throughput is 55 Mb/s on the x86 target. This is very close to the sequential throughput of t_5^{Rx} on the same target (55.42 Mb/s). Knowing that this task is the limiting factor in the pipeline, it demonstrates the efficiency of the proposed multi-threaded implementation.

The throughputs obtained on the ARM target are lower than on the x86 CPUs (by a factor of ≈ 2 when running in parallel). It can be explained by the limited mono-core performance of the ThunderX2 architecture: the frequency is lower (2.0 GHz versus 2.7 GHz) and the SIMD width is smaller (128-bit in NEON versus 512-bit in AVX-512F).

IV. RELATED WORKS

Some other works are focusing on SDR implementations of a DVB-S2 transceiver. To the best of our knowledge, the competitive existing projects are discussed thereafter.

gr-dvbs2rx [1] is an open source extension to GNU Radio. The project sounds promising but lacks efficiency. Its main maintainer affirms that the receiver is not yet able to meet the satellite real time constraints on a Xeon™ Gold/Platinum processor. The use of a dedicated GPU or an FPGA for the LDPC decoding is advised.

leansdr [7] is a standalone open source project. The project creation was motivated to reach higher receiver throughput than GNU Radio at the cost of decoding performance degradation. For instance, a low complexity LDPC bit-flipping decoder is chosen. At the time of the writing, the project does not support multi-threading.

Grayver and Utter recently published a paper [2] in which they succeed to build a 10 Gb/s DVB-S2 receiver on a cluster of server-class CPUs. On a comparable CPU, their work is able to double or even triple the throughput of our implementation (approximately 185 Mb/s on 20 cores). This is mainly due to new algorithmic improvements in the synchronization tasks and the use of a very fast LDPC decoder [8], [9]. They are able to express more parallelism in the *Synchronizer Timing* task (t_5^{Rx}). In our work, we could have split t_5^{Rx} in multiple pipeline stages to increase its throughput but we preferred to stop the optimization process since we met the industrial constraints. One may note that Grayver and Utter's work focuses on a single DVB-S2 MODCOD (8-PSK, $N = 64800$ and $R = 1/2$) and the achieved decoding performances are not reported.

V. CONCLUSION

This paper deals with the software implementation of real-time DVB-S2 transceiver. For this purpose, USRP modules were combined with multicore and SIMD CPUs. The implementations were done thanks to the AFF3CT toolbox. Thus, some components are directly from the AFF3CT library and others such as the synchronization functions have been described. Experiment results show the performance but also the flexibility and the portability of the transceiver. An obvious strength of this work is also its collaborative context and the code publicly released.

REFERENCES

[1] R. Economos. (2018) gr-dvbs2rx: GNU Radio Extensions for the DVB-S2 and DVB-T2 Standards. [Online]. Available: https://github.com/drmpeg/gr-dvbs2rx

[2] E. Grayver and A. Utter, "Extreme Software Defined Radio – GHz in Real Time," in *Aerospace Conference*. IEEE, 2020.

[3] ETSI, "EN 302 307 - Digital Video Broadcasting (DVB)," 2005. [Online]. Available: https://www.etsi.org/deliver/etsi_en/302300_302399/302307/01.02.01_60/en_302307v010201p.pdf

[4] A. Cassagne *et al.*, "AFF3CT: A Fast Forward Error Correction Toolbox!" *Elsevier SoftwareX*, vol. 10, 2019.

[5] ——, "MIPP: A Portable C++ SIMD Wrapper and its use for Error Correction Coding in 5G Standard," in *WPMVP*. ACM, 2018.

[6] F. Broquedis *et al.*, "hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications," in *PDP*. IEEE, 2010.

[7] pabr. (2016) leansdr: Lightweight, Portable Software-defined Radio. [Online]. Available: https://github.com/pabr/leansdr

[8] B. Le Gal and C. Jégo, "High-Throughput Multi-Core LDPC Decoders Based on x86 Processor," *TPDS*, vol. 27, no. 5, pp. 1373–1386, 2016.

[9] E. Grayver, "Scaling the Fast x86 DVB-S2 Decoder to 1 Gbps," in *Aerospace Conference*. IEEE, 2019.