Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

A DSEL for High Throughput and Low Latency Software-Defined Radio on Multicore CPUs

Adrien Cassagne*[1] | Romain Tajan[2] | Olivier Aumage[3] | Camille Leroux[2] | Denis Barthou[3] | Christophe Jégo[2]

[1]Sorbonne Université, CNRS, LIP6, F-75005, Paris, France
[2]IMS Laboratory, UMR CNRS 5218, Bordeaux INP, University of Bordeaux, Talence, France
[3]Inria, Bordeaux Institute of Technology, LaBRI/CNRS, Bordeaux, France

Correspondence
*Adrien Cassagne, LIP6 lab. (ALSOC team), Sorbonne University. Email: adrien.cassagne@lip6.fr

Summary

This article presents a new Domain Specific Embedded Language (DSEL) dedicated to Software-Defined Radio (SDR). From a set of carefully designed components, it enables to build efficient software digital communication systems, able to take advantage of the parallelism of modern processor architectures, in a straightforward and safe manner for the programmer. In particular, proposed DSEL enables the combination of pipelining and sequence duplication techniques to extract both temporal and spatial parallelism from digital communication systems. We leverage the DSEL capabilities on a real use case: a fully digital transceiver for the widely used DVB-S2 standard designed entirely in software. Through evaluation, we show how proposed software DVB-S2 transceiver is able to get the most from modern, high-end multicore CPU targets.

KEYWORDS:
DSEL, SDR, Multicore CPUs, Pipeline, Real-time system, DVB-S2 transceiver

1 | INTRODUCTION

Digital communication systems are traditionally implemented onto dedicated hardware (ASIC) to achieve high throughputs, low latencies and energy efficiency. However, hardware implementations suffer from a long time to market, are expensive and specific by nature[1,2]. New communication standards such as the 5G are coming with large specifications and numerous possible configurations[3]. Connecting objects that exchange small amounts of data at low rates will live together with 4K video streaming for mobile phone games requiring high throughputs and low latencies[4].

To meet such diverse specifications, transceivers have to be able to adapt quickly to new configurations. Flexible, reconfigurable and programmable solutions are thus increasingly required, fueling a growing interest for the Software-Defined Radio (SDR). It consists in processing both the Physical (PHY) and Medium Access Control (MAC) layers in software[5], rather than in hardware. Shorter time to market, lower design costs, ability to be updated, to support and to interoperate with new protocols are its main advantages[6].

SDR can be implemented on various targets such as Field Programmable Gate Arrays (FPGAs)[7,8,9,10,11,12], Digital Signal Processors (DSPs)[13,14,10] or General Purpose Processors (GPPs)[15,16,17,18]. Many SDR elementary blocks have been optimized for Intel® and ARM® CPUs. High throughput results have been achieved on GPUs[19,20,21,22,23]; latency results are is still too high however to meet real time constraints and to compete with CPU implementations[24,25,26,22,27,28,29,30,31,32,33]. This is mainly due to data transfers between the host (CPUs) and the device (GPUs), and to the nature of GPU designs, which are not optimized for latency efficiency. In this paper, we focus on the execution of SDR on multicore general purpose CPUs.

Digital communication systems can be refined so that the transmitter and the receiver parts are decomposed into several processing blocks connected in a directed graph. This matches the dataflow model[34,35]: Blocks are filters and links between blocks are data exchanges. Specific dataflow models such as the synchronous dataflow[36] and the cyclo-static dataflow[37,38] allow the expression of a static schedule for the graph[39]. SDR however requires a parallel task graph between stateful tasks and a dynamic schedule due to early exits, conditionals and loop iterations. Maximizing throughput is the main objective, keeping latency as low as possible is a secondary objective. This constrains the time taken by data movements and requires optimizing for parallelism. For a SDR operating in antennas or transceivers, memory footprint is not an issue and all tasks run on CPU cores.

This paper is an extension of a previous work[40] in which we presented a complete, real-life example, digital communication system (a DVB-S2 transceiver) running on both x86 (Intel®) and ARM (Cavium®) CPUs, that meets real-time requirements for ground-satellite transceivers. This paper includes the DVB-S2 software transceiver as well as the following new contributions:

- A DSEL based on C++ to build parallel dataflow graphs for SDR signal processing, supporting loops (including nested and input-dependent ones), conditionals, early exits, pipeline and fork/join parallelism;
- A set of micro-benchmarks to analyze the time taken by the different constructs;
- In-depth insights to understand the performance and the genericity of the proposed DVB-S2 transceiver (for instance, two pipeline implementations are compared and its implementation with the proposed DSEL is shown);
- A comprehensive and "as fair as possible" comparison with State-of-the-Art DVB-S2 software transceivers.

Section 2 discusses related works. The proposed DSEL is presented in Section 3. Section 4 details scheduling and parallelism supports. Section 5 experiments with a DVB-S2 implementation built on the DSEL.

2 | RELATED WORKS

Many languages dedicated to streaming applications have been introduced[41,42,43,44,45,46,47,48]. These languages are often variants of the cyclo-static dataflow model and propose automatic parallelization techniques such as pipelining and forks/joins.

The concepts presented in StreamIt[41] are very close to the ones implemented in the proposed DSEL. However, StreamIt does not provide dynamic loops and conditions. *FeedbackLoop* and *SplitJoin* are closer but their scheduling is static. On the other hand, some concepts like a dedicated control messaging system are not implemented in the proposed DSEL. Even if it is still possible to embed control messages in the data, this is less advanced than what StreamIt does.

In Dardaillon et al. work[49], a full compilation chain for SDR, based on LLVM on heterogeneous MPSoCs. This is promising but it differs from our approach. Indeed, we chose to integrate our language into C++, making the DSEL compatible with any C++11 compilers. Works are also tackling OS and hardware aspects of SDR[50,51,52]. However, the studied SDR systems are much simpler than those addressed in this paper.

Few solutions specifically target SDR sub-domain so far. GNU Radio[53] is the most famous one. It is open source and largely adopted by the community. It comes bundled with a large variety of digital communication techniques used in real life systems. GNU Radio models digital communication systems at the symbol and frame level. The philosophy is close to typical algorithm descriptions from the signal processing literature, which enables fast prototyping of new digital processing algorithms. The last version of GNU Radio (3.9) can take advantage of multi-core CPUs. One thread is spawned per block and the scheduling is directly managed by the operating system. While sufficient on uniform memory access (UMA) architectures[54], this does not take into account non uniform memory access (NUMA) architectures with many cores. A drawback of assigning a block per thread is that the designed SDR system is strongly linked to the parallelism strategy. Depending on the CPU architecture, it can be necessary to change the parallelism strategy while keeping the same SDR system description. GNU Radio designers are currently working on a proof of concept scheduler (newsched) for the future GNU Radio version 4[55]. They introduced the concept of workers that can execute more than one block on a physical core. To the best of our knowledge, this new version of GNU Radio breaks the compatibility with the existing systems designed with GNU Radio and is not yet fully implemented. However, this new version goes in the same direction as what we propose and we hope that some contributions of this paper could help the GNU Radio project. To the best of our knowledge, GNU Radio does not implement the duplication mechanism presented in Section 3.3 and we show this is a key mechanism for high throughputs and scalability. Besides, a new construct in the next section (cf. the switcher module in Section 3.1) allows the design of loops and conditions for SDR systems, not yet supported by GNU Radio where only static directed acyclic graph can be managed.

Some other works are focusing particularly on an SDR implementation for a DVB-S2 transceiver. Hereafter are the projects we have identified:

- **leansdr**. A standalone open source project[56]. A low-density parity-check (LDPC) bit-flipping decoder[57] is chosen. The project does not support multi-threading.
- **gr-dvbs2rx**. An open source out-of-tree module[58] for GNU Radio. One of the motivation of this project is to increase the throughput compared to leansdr. The project is open-source and we were able to perform a fair comparison. The results are presented in Section 5.5.
- **Grayver and Utter**. In a recently published paper[18], they succeed in building a 10 Gb/s DVB-S2 receiver on a cluster of server-class CPUs. The implementation is closed source, making fair comparisons difficult.

3 | DESCRIPTION OF THE PROPOSED DOMAIN SPECIFIC EMBEDDED LANGUAGE

This section introduces a DSEL working on *sets of symbols* (aka frames). It implements a form of the dataflow model, single rate, tailored to the relevant characteristics of digital communication chains with channel coding. The language defines *elementary* and *parallel* components.

3.1 | Elementary Components

Four elementary components are defined: *sequence*, *module*, *task* and *socket*. The *task* is the fundamental component. It can be an encoder, a decoder or a modulator for instance and is a single-threaded code function. It is designated as *filter* in the standard dataflow model. Though unlike a dataflow filter, a task can have an internal state and a private memory to store temporary data. Additionally, a set of tasks can share a common internal/private memory. In that case, multiple tasks are grouped into a single *module*. The main problem with internal memory is that tasks cannot be executed safely by several threads in parallel because of data races. However, in many cases the expression of one task or a set of tasks can be simplified by allowing stateful tasks and modules.

A task can consume and produce public data through the input and/or output *sockets* it exposes. Connecting the sockets of different tasks is called *binding*. An input socket can only be bound to one output socket, while an output socket can be bound to multiple input sockets. A task can only be executed once all its input sockets are bound.

Tasks can be grouped into a *sequence*. A sequence corresponds to a static schedule of tasks. To create a sequence, the designer specifies the first tasks and the last tasks to execute. Then the connected tasks are analyzed and a sequence object is built. The analysis is a depth-first traversal of the task graph and independent tasks are ordered according to the binding order of their inputs. The principle is to add a task to the array of function pointers when all the input sockets are visited in the depth first traversal of the tasks graph. After that, the output sockets of the current task are followed to reach new tasks. The order in which the tasks have been traversed is memorized in the sequence. When the designer calls the `exec` method on a sequence, the tasks are executed successively according to this statically scheduled order.

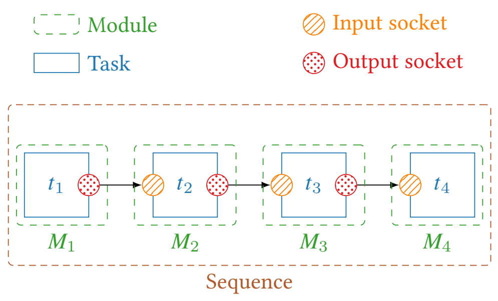


FIGURE 1 Simple chain sequence.

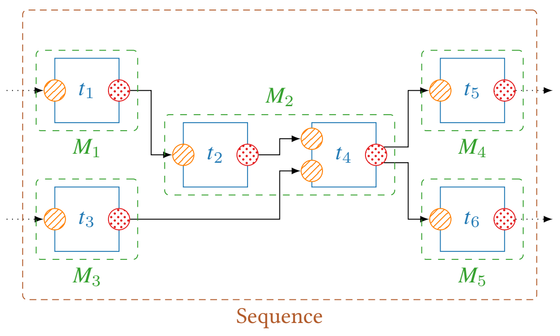


FIGURE 2 Sequence with multiple first and last tasks.

```
 1  // 1) create the module objects
 2  M1 m1(); M2 m2(); M3 m3(); M4 m4();
 3
 4  // 2) bind the tasks
 5  m2["t2::in"] = m1["t1::out"];
 6  m3["t3::in"] = m2["t2::out"];
 7  m4["t4::in"] = m3["t3::out"];
 8
 9  // 3) create the sequence (stop
10  //    automatically at t4 task)
11  Sequence seq(m1["t1"]);
12
13  // 4) execute the sequence (tasks
14  //    graph is executed 100 times)
15  unsigned int exe_counter = 0;
16  seq.exec([&exe_counter]() {
17     return ++exe_counter >= 100;
18  });
```

LISTING 1: Source code of the simple chain sequence presented in Fig. 1.

```
 1  // 1) create the module objects
 2  M1 m1(); /* ... */ M7 m7();
 3
 4  std::vector<TYPE> some_data(SIZE);
 5
 6  // 2) bind the tasks
 7  m1["t1::in" ] = some_data;
 8  m3["t3::in" ] = some_data;
 9
10  m2["t2::in" ] = m1["t1::out"];
11  m2["t4::in1"] = m2["t2::out"];
12  m2["t4::in2"] = m3["t3::out"];
13  m4["t5::in" ] = m2["t4::out"];
14  m5["t6::in" ] = m2["t4::out"];
15
16  m6["t7::in" ] = m4["t5::out"];
17  m7["t8::in" ] = m5["t6::out"];
18
19  // 3) create the sequence
20  Sequence seq(
21    { m1["t1"], m3["t3"] }, // first tasks
22    { m4["t5"], m5["t6"] }); // last tasks
23
24  // 4) execute the sequence (no stop)
25  seq.exec([]() { return false; });
```

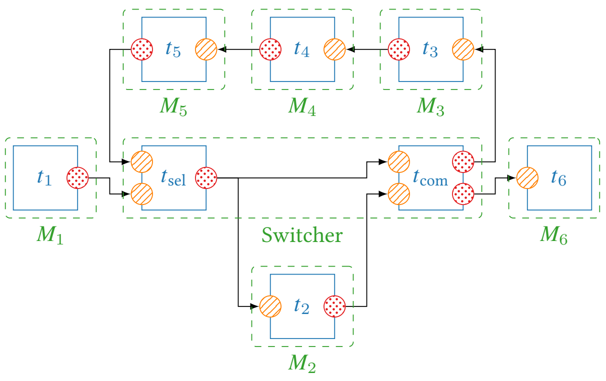
LISTING 2: Source code of the sequence with multiple first and last tasks presented in Fig. 2.

Fig. 1 and Fig. 2 show two examples of task sequences. Fig. 1 is a simple chain of tasks. The designer only needs to specify its first task (t_1); the sequence analysis then follows the binding until the last task (t_4). The corresponding source code is given in Listing 1. Here we suppose that modules $M_{1:4}$ have been previously implemented. Lines 5-7 perform the socket bindings, line 11 shows the creation of the sequence from the t_1 task and line 16 is the sequence execution. In Fig. 2, bound tasks exist before and after the current sequence, which also has two *first* tasks (t_1 and t_3) and two *last* tasks (t_5 and t_6). In this case, the designer has to explicitly specify that t_1 and t_3 are first tasks. If t_1 is sequentially defined before t_3 then t_1 will be executed first and t_3 after. The analysis starts from t_1 and continue to traverse new tasks if possible. In this example, t_2 can be executed directly after t_1, but t_4 cannot because it depends on t_3. So the analysis stops after t_2 and then restarts from t_3. Actually, the index i of the t_i task represents the execution order. The t_5 and t_6 last tasks have to be explicitly specified because their output sockets are bound: the analysis cannot guess the end of the sequence. The corresponding source code is given in Listing 2. Lines 20-22 show the creation of the sequence from two first tasks (t_1 and t_3) to two last tasks (t_5 and t_6). The first two tasks (t_1 and t_3) have one input socket each. As they are the first tasks, they will be executed in the sequence in all cases. In the example, their input sockets are bound to a C++ *vector* (lines 4-8). They could also have been bound to other tasks or to a C array. At the end of the bindings (line 16-17), t_7 and t_8 tasks are bound to t_5 and t_6 tasks, resp. However, t_7 and t_8 tasks are not executed in the sequence.

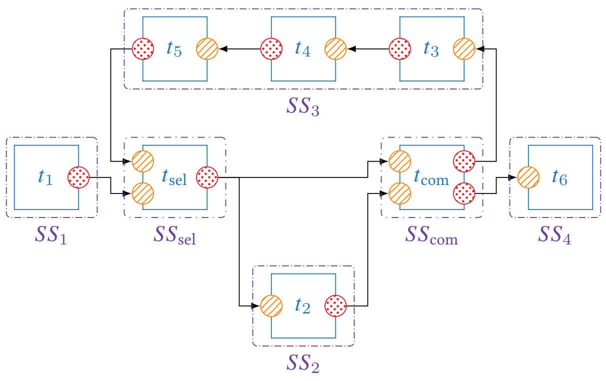
In targeted SDR applications, processing is continuously repeated on batches of frames as long as the system is on. A sequence is thus executed in a loop. When the last sequence task is executed, the next task is the first one on the next frame. The designer can control whether the sequence should restart by supplying a *condition function* to the sequence `exec` method. The boolean returned by the function conditions whether the sequence is repeated. For instance, in Listing 1, the *condition function* is given as a lambda function which increments a counter. When this counter reaches 100, the condition line 17 is verified and the sequence ends. On the other hand, in Listing 2, the lambda function always returns *false* and the sequence is executed indefinitely. A task of a sequence may also raise an abort exception upon some condition, to immediately stop the current sequence execution and start the first task of the sequence with the next frame. In Fig. 1 if the t_3 task raises the abort exception then the next executed task is t_1 instead of t_4.

Some digital communication systems include schemes that require a loop or a conditional. A sequence of tasks is executed one or more times depending on a *condition* task (or a *control* task). To build loops and conditionals, we introduce a switcher module composed of two control flow tasks. The *select* task selects one among several exclusive input paths. The *commute* task creates two or more exclusive output paths.

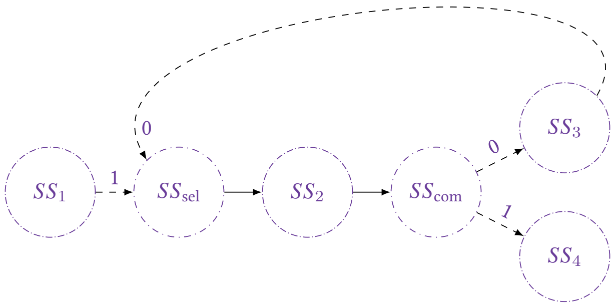
Fig. 3 illustrates a loop. To build a loop structure (a while loop in the example), the *select* and *commute* tasks (in the given order) of a common switcher module are used (see Fig. 3a). By convention, in a switcher module, the selected path is initialized



(a) Decomposition of the loop in modules.



(b) Decomposition of the loop in sub-sequences.



(c) Execution graph.

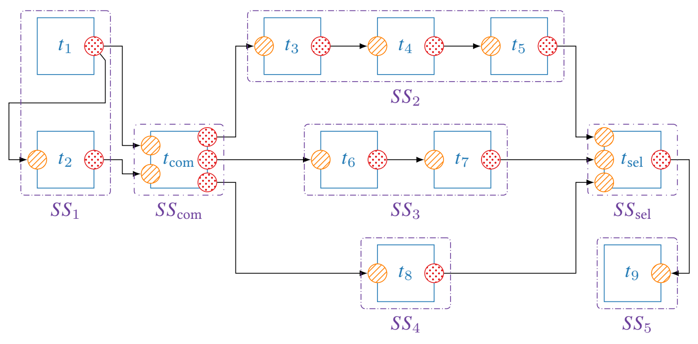
FIGURE 3 Example of a sequence with a while loop.

Algorithm 1 Pseudo code of the loop (corresponding to Fig. 3)

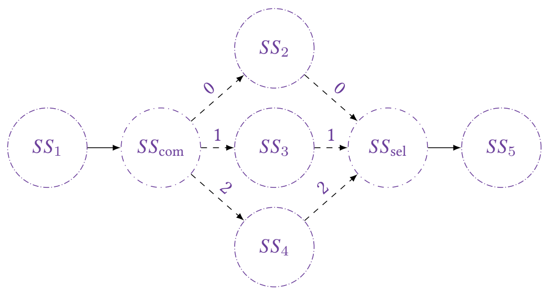
```
execute $SS_1$;
**while** execute $SS_2$ **and not** $t_{com}$.in$_2$:
    execute $SS_3$;
execute $SS_4$;
```

to the highest possible path (here 1). So, at the first t_{sel} execution, the t_1 output will be selected. Then the t_2 control task will send 0 or 1 as a control socket to t_{com}. Here the t_2 loop control task is based on the t_{sel} output socket. As a consequence, the path selection (0 or 1) is dynamic and depends on the runtime data. It is also possible to model the `for` loop behavior by ignoring the t_2 input data and by adding an internal state to M_2, namely the loop counter. If t_{com} receives a 0, then the internal path of the switcher will be 0. Then t_3, t_4 and t_5 tasks will be executed and t_{sel} will select the t_5 output instead of the t_1 output, and so on. Fig. 3b shows how the tasks are regrouped into sub-sequences. It enables to build the execution graph illustrated in Fig. 3c. The corresponding pseudo code of the presented loop is shown in Alg. 1. One can note that in the proposed DSEL there is no limitation to include nested loops schemes. The loop pattern is common in iterative demodulation/decoding. This is why it is required in a DSEL dedicated to SDR. As an exception rule in the graph construction, the *select* task is added to the graph when its last input socket is visited (while all the other tasks require all input sockets to be visited).

Fig. 4 illustrates a `switch` structure. The same switcher module presented in while loop structures is necessary. The number of output/input sockets in resp. t_{com}/t_{sel} tasks is 3 instead of 2 in the while loop example. Also, the position of these tasks has been swapped, in the current example t_{com} is executed before t_{sel}. t_2 is a control task that depends on the output of t_1. The t_2 task output can be 0, 1 or 2. The switch exclusive path is determined dynamically depending on the runtime data. Fig. 4a shows the decomposition of the tasks in sub-sequences and Fig. 4b presents the resulting execution graph. Alg. 2 gives the corresponding pseudo code. The switch pattern is useful in many SDR contexts. For instance, depending on the signal to noise ratio (SNR), the receiver can select a different path adapted to the signal quality.



(a) Decomposition of the switch in sub-sequences.



(b) Execution graph.

FIGURE 4 Example of a sequence with a switch.

Algorithm 2 Pseudo code of the switch (corresponding to Fig. 4)

```
execute $SS_1$;
**switch** $t_{com}.in_2$:
    **case** 0: execute $SS_2$;
    **case** 1: execute $SS_3$;
    **case** 2: execute $SS_4$;
execute $SS_5$;
```

```
M1 m1(); /* ... */ M6 m6();
Switcher sw(2); // 2 exclusive paths

sw["tsel::in2"] = m1[  "t1::out" ];
m2[  "t2::in" ] = sw["tsel::out" ];
sw["tcom::in1"] = sw["tsel::out" ];
sw["tcom::in2"] = m2[  "t2::out" ];
// sub-seq. 3, executed if tcom::in2 = 0
m3[  "t3::in" ] = sw["tcom::out1"];
m4[  "t4::in" ] = m3[  "t3::out" ];
m5[  "t5::in" ] = m4[  "t4::out" ];
sw["tsel::in1"] = m5[  "t5::out" ];
// sub-seq. 4, executed if tcom::in2 = 1
m6[  "t6::in" ] = sw["tcom::out2"];

Sequence seq(m1["t1"]);
seq.exec([]() { return false; });
```

LISTING 3: Source code of the sequence with a while loop presented in Fig. 3.

```
M1 m1(); /* ... */ M9 m9();
Switcher sw(3); // 3 exclusive paths

sw["tcom::in1"] = m1[  "t1::out" ];
m2[  "t2::in" ] = m1[  "t1::out" ];
sw["tcom::in2"] = m2[  "t2::out" ];
// sub-seq. 2, executed if tcom::in2 = 0
m3[  "t3::in" ] = sw["tcom::out1"];
m4[  "t4::in" ] = m3[  "t3::out" ];
m5[  "t5::in" ] = m4[  "t4::out" ];
// sub-seq. 3, executed if tcom::in2 = 1
m6[  "t6::in" ] = sw["tcom::out1"];
m7[  "t7::in" ] = m6[  "t6::out" ];
// sub-seq. 4, executed if tcom::in2 = 2
m8[  "t8::in" ] = sw["tcom::out1"];
// merge exclusive paths
sw["tsel::in1"] = m5[  "t5::out" ];
sw["tsel::in2"] = m7[  "t7::out" ];
sw["tsel::in3"] = m8[  "t8::out" ];
// last task binding
m6[  "t6::in" ] = sw["tsel::out" ];

Sequence seq(m1["t1"]);
seq.exec([]() { return false; });
```

LISTING 4: Source code of the sequence with a switch presented in Fig. 4.

Listing 3 and Listing 4 present the source codes corresponding to the loop example in Fig. 3 and the switch example in Fig. 4, respectively. One may note that the proposed DSEL does not introduce new C++ keywords, it is only based on tasks binding. Thus, it can run on any C++ (version 11) compilers. In the previous listings, the *Switcher* module is a keyword of the DSEL (but not a C++ keyword). When the sequence is created, the task graph is parsed and the `sw` object is processed as an exception. The C++ runtime *dynamic cast* feature is used to recognized objects of *Switcher* class. Thus, the proposed DSEL can be seen as an interpreted language. The code corresponding to the tasks graph is generated only once when the sequence object is created (line 16 in Listing 3 and line 23 in Listing 4).

```cpp
 1 // create a stateless module
 2 Stateless min32();
 3 // set module name
 4 min32.set_name("Minimum32");
 5 // create a task for the 'min32' module
 6 Task &t = min32.create_tsk("find_min");
 7 // create in/out sockets for the task
 8 size_t si =
 9   min32.create_sck_in<int>(t, "in", 32);
10 size_t so =
11   min32.create_sck_out<int>(t, "out", 1);
12 // define the code to execute when the
13 // 'find_min' task is called
14 min32.create_codelet(t,
15   [si, so](Module &m, Task &tsk) -> int {
16   // get in/out data pointers
17   const int* pi = tsk[si].get_dataptr();
18   int* po = tsk[so].get_dataptr();
19   // compute the minimum of 32 elements
20   *po = pi[0];
21   for (int i = 1; i < 32; i++)
22     if (*po > pi[i])
23       *po = pi[i];
24   // return 'SUCCESS' code
25   return status_t::SUCCESS;
26 });
```

LISTING 5: Definition and implementation of a stateless task that finds the minimum in a 32 elements array.

```cpp
 1 class Counter : public Module {
 2 private: // inner data => stateful
 3 int cnt;
 4 public: // constructor
 5 Counter() : Module(), cnt(0) {
 6   this->set_name("Counter");
 7   Task &t = this->create_tsk("get_val");
 8   size_t so =
 9     this->create_sck_out<int>(t, "o", 1);
10   this->create_codelet(t,
11     [so](Module &m, Task &tsk) -> int {
12     int* po = tsk[so].get_dataptr();
13     // cast 'm' into 'Counter' class type
14     Counter &mc =
15       static_cast<Counter&>(m);
16     // write value in the output socket
17     *po = mc.get_value();
18     // increment the counter
19     mc.increment();
20     return status_t::SUCCESS;
21   });
22 }
23 protected: // methods
24 int get_value() { return this->cnt; }
25 void increment() { this->cnt++; }
26 };
```

LISTING 6: Definition and implementation of a stateful task that increments a counter.

Listing 5 shows how to define and implement a stateless task in the proposed DSEL. First, a *Stateless* class is instantiated (line 2), then a task is declared (`find_min` line 6) with its input/output sockets (line 8-9 and 10-11). To create a socket, four parameters are required: its datatype (given as a template parameter), its associated task, its name, and its size. Finally, a "codelet" function need to be set (lines 14-26). This codelet will be called when the task will be triggered. In the Listing 5, the codelet is given as a lambda function with two parameters: a module and a task. In the case of a stateless task, the module is never used. However, the task is used to get data pointers associated to each socket (lines 17-18). Then the code to find the minimum element is written (lines 20-23). In addition, a codelet returns a status integer that can be used by other tasks. The return status integer is implemented as a particular output socket.

Listing 6 shows how to define and implement a stateful task in the proposed DSEL. The state of the task is represented by a member of the `Counter` class, namely `cnt` (line 3). The `Counter` class inherits from the *Module* abstract class. The latest defines and implements specific methods required to declare tasks and sockets. In the example, the `get_val` task and its output socket are defined in the constructor of the `Counter` class (lines 7-21). In the case of a stateful task, the codelet needs the module parameter to read/write the inner data. In Listing 6, this is achieved by calling the `get_value` and `increment` methods on the `mc` module object (lines 16-19).

One may note that stateless and stateful task definitions are well suited to wrap existing C/C++ code. Indeed, the functions have to be described with tasks (`create_tsk` function) and the corresponding input/output parameters have to be described with sockets (`create_sck_in` and `create_sck_out` functions). Then, these functions need to be called in the codelet.

3.2 | Performance Evaluation on Micro-benchmarks

In this section, an estimation of the DSEL overhead is measured from four micro-benchmarks: a simple chain (see Fig. 1) denoted MB_1, a single `for` loop (Fig. 3) denoted MB_2, a system of two nested `for` loops denoted MB_3, and a system with a `switch` (Fig. 4) denoted MB_4. These micro-benchmarks are representative of real-world SDR application patterns and they are available online[†]. In MB_1, MB_2 and MB_3 three computational tasks are chained. In MB_2, the loop performs 10 iterations. In

[†]Micro-benchmarks source code: https://github.com/aff3ct/aff3ct-core/tree/development/tests)

MB_3, the inner loop performs 5 iterations, the outer loop performs 2 iterations. In MB_4, three computational tasks are chained in the first path, two in the second path and a single in the last path. Moreover, an iterate task is configured to perform a cyclic path selection (0,1,2,0,1,2,...). In each computational task, an active wait of the same amount of time is performed. Four types of tasks are used: *computational* tasks $\mathcal{C}$, *select* and *commute* switcher module tasks $\mathcal{S}_{\text{sel}}$ and $\mathcal{S}_{\text{com}}$ resp., and *iterate* tasks $\mathcal{I}$ to determine paths in loops and switches ($\mathcal{I}$ = control task).

Evaluations ran on a single core of an Intel® Core™ i5-8250U @ 1.60 GHz. The *Turbo Boost* mode has been disabled. This processor has a 15-Watt TDP that matches embedded system constraints. The GNU C++ compiler version 9.3 (on Linux Ubuntu 20.04 operating system) has been used with the following flags: `-O3 -march=native`.

Though duration of a $\mathcal{C}$ task is controlled by the programmer, we measured a constant 135 ns overhead due to the DSEL and to the system call behind the `std::chrono::steady_clock::now()` function. We measured $\mathcal{S}_{\text{sel}}$ tasks around 60 ns, $\mathcal{S}_{\text{com}}$ tasks around 80 ns, and $\mathcal{I}$ tasks around 70 ns. Later on, $\mathcal{S}_{\text{sel}}$, $\mathcal{S}_{\text{com}}$ and $\mathcal{I}$ tasks are reported as overhead. both $\mathcal{S}_{\text{sel}}$ and $\mathcal{S}_{\text{com}}$ tasks are copy-less, thus for a given configuration, their execution time is constant.

TABLE 1 Execution of 1 125 000 $\mathcal{C}$ tasks of 4 μs each. Theoretical execution time is 4500 ms for each micro-benchmark.

			Overhead								
			$\mathcal{C}$ tasks		$\mathcal{S}_{\text{sel}}$ tasks		$\mathcal{S}_{\text{com}}$ tasks		$\mathcal{I}$ tasks		Other
Label	Seq. exec.	Run time (ms)	Exec.	Time (ms)	Exec.	Time (ms)	Exec.	Time (ms)	Exec.	Time (ms)	Time (ms)
MB_1	375000	4656.45	1125000	151.86	–	–	–	–	–	–	4.59
MB_2	37500	4744.08	1125000	151.86	412500	24.75	412500	33.00	412500	28.88	5.59
MB_3	37500	4777.03	1125000	151.86	562500	33.75	562500	45.00	562500	39.38	7.04
MB_4	562500	4784.88	1125000	151.86	562500	33.75	562500	45.00	562500	39.38	14.89

Tab. 1 reports the execution time of 1 125 000 $\mathcal{C}$ tasks for each case. Column *Seq. exec.* gives the number of sequence executions required to run 1 125 000 $\mathcal{C}$ tasks. Theoretical time is computed directly from the number of $\mathcal{C}$ tasks and the duration of the active waiting in each task: $\mathcal{T}_{\text{theoretical}} = 1125000 \times 4\ \mu\text{s} = 4500$ ms. *Run time* column reports the measured execution time. Remaining columns report task counts and overheads per task types. Last column *Other* reports the residual time that does not come from the tasks execution. In each benchmark, a stop condition is evaluated at the end of the sequence. The condition checks that the current number of executions is lower than the one given in the *Seq. exec.* column. This comes with an extra cost because of an additional function call for each sequence execution. This is also why the execution time of MB_4 is higher than MB_3, there is significantly more sequence executions in MB_4.

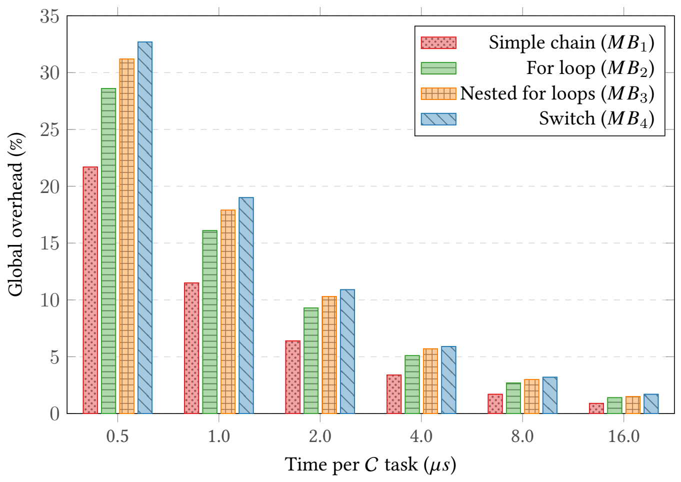


FIGURE 5 Global overhead of the proposed DSEL on 4 micro-benchmarks.

Fig. 5 shows the overhead depending on the granularity of the C tasks. It results that from 4 μs tasks, the proposed DSEL has an acceptable overhead. For tasks longer than 4 μs the overhead is negligible. This shows that the proposed DSEL matches the low latency requirements of SDR systems.

3.3 | Parallel Components

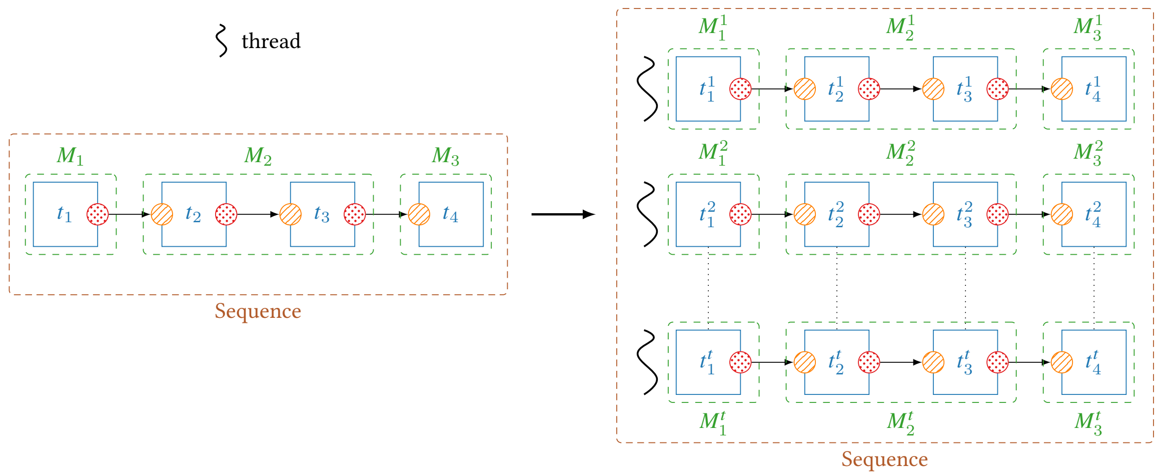


FIGURE 6 Sequence duplication for multi-threaded execution. The modules are duplicated with their tasks and data.

A sequence can be duplicated, to let several threads execute it in parallel, see Fig. 6. The number t of duplicates is a parameter of its constructor. There is no synchronization between sequence duplicates. Each threaded sequence can be executed on one dedicated core and the public data transfers remain on this core for the data reuse in the caches. By default, modules have no duplication mechanism (see next section).

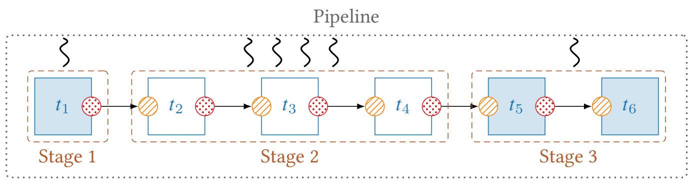
In some particular cases such as in the signal synchronization processing, the tasks can have a dependency on themselves. It is then impossible to duplicate the sequence because of the sequential nature of the tasks. To overcome this issue, the well-known pipelining strategy can be applied to increase the sequence throughput up to the slowest task throughput. The proposed DSEL comes with a specific *pipeline* component to this purpose. The pipeline takes multiple sequences as input. Each sequence of the pipeline is called a *stage*, run on one thread. For instance, a 4-stage pipeline creates 4 threads. A pipeline stage can be combined with the sequence duplication strategy. It means that there are nested threads in the current stage thread. Pipelining comes with an extra synchronization cost between the stage threads, implementation details are discussed in the next section.

4 | AUTOMATED PARALLELIZATION TECHNIQUES

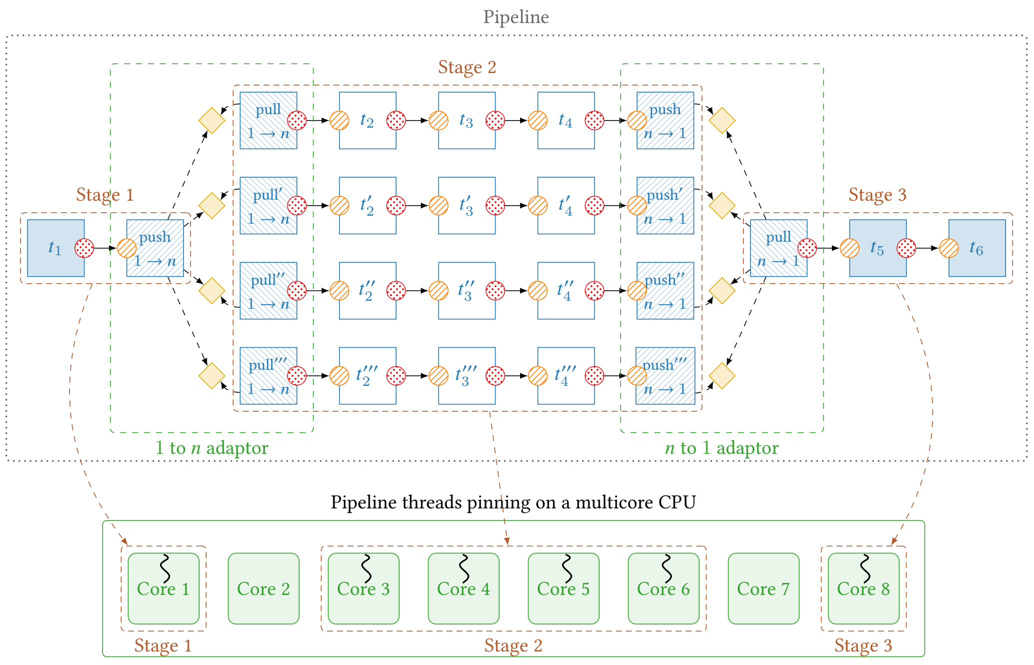
4.1 | Sequence Duplication

In a fully dataflow-compliant model, there is no need to duplicate the sequence because a stateless task is always thread-safe. In the proposed DSEL with stateful tasks, a *clone* method is defined for each module to deal with internal state and private memory (stored in the module). The *clone* method is polymorphic and defined in the *Module* abstract class. It relies on the implicit copy constructors and a *deep copy* protected method (overridable). It is the responsibility of the *ModuleImpl* developer to correctly override the *deep copy* method and to make sure the duplication is valid for this module. The *deep copy* method deals with pointer and reference members. If the pointer/reference members are read-only (`const`), then the implicit copy constructor copies the memory addresses automatically. When the current *ModuleImpl* class owns one ore more writable references, the module cannot be cloned and its tasks are sequential. However, for a writable pointer member, the developer can explicitly allocate a new pointer in the *deep copy* method.

4.2 | Pipeline



(a) Description of a pipeline: tasks creation, tasks binding and sequences/stages definition (with the corresponding number of threads).



(b) Automatic parallelization of a pipeline description: sequence duplications, 1 to n and n to 1 adaptors creation and binding.

FIGURE 7 Example of a pipeline description and the associate transformation with adaptors.

```
// 1) creation of the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4(); M5 m5(); M6 m6();
// 2) binding of the tasks
m2["t2::in"]=m1["t1::out"]; m3["t3::in"]=m2["t2::out"]; m4["t4::in"]=m3["t3::out"];
m5["t5::in"]=m4["t4::out"]; m6["t6::in"]=m5["t5::out"];
// 3) creation of the pipeline (= sequences and pipeline analyses)
runtime::Pipeline pip(
  m1["t1"], // first task of the graph (for valisation purpose)
  { { { m1["t1"] }, { m1["t1"] } },    // first & last tasks of stage 1
    { { m2["t2"] }, { m4["t4"] } },    // first & last tasks of stage 2
    { { m5["t5"] }, { m6["t6"] } }, }, // first & last tasks of stage 3
  { 1, 4, 1 }, // number of threads per stage
  { {          1 },      // pin thread  '1'   of stage 1 to core  '1'
    { 3, 4, 5, 6 },      // pin threads '1-4' of stage 2 to cores '3-6'
    {          8 }, }); // pin thread  '1'   of stage 3 to core  '8'
// 4) execution of the pipeline, it is indefinitely executed in loop
pip.exec([]() { return false; });
```

LISTING 7: C++ DSEL source code of the pipeline described in Fig. 7.

In this section, the pipeline implementation is illustrated through a simple example. Fig. 7 shows the difference between a pipeline description (see Fig. 7a) and its actual instantiation (see Fig. 7b). In Fig. 7 we suppose that the t_1, t_5 and t_6 tasks cannot be duplicated (plain boxes). The designer knows that the execution time of the t_1 task is higher than the cumulated execution time of tasks t_5 and t_6. We assume that the cumulated execution time of t_2, t_3 and t_4 is approximatively four times higher than t_1. This knowledge motivates the splitting of the stages 1, 2 and 3. There is no need to split the t_5 and t_6 tasks in two stages because the overall throughput is limited by the slowest stage (t_1 here). Stage 2 is duplicated four times to increase its throughput by four as we know that its latency is approximatively four times that of Stage 1. In general, a preliminary profiling phase of the sequential code is required to guide the pipeline strategy. In the proposed DSEL, each task can be automatically profiled: its duration is computed with the C++11 `chrono` standard library. For a given task, the API returns the average latency, the minimum latency and the maximum latency. In this work, the following strategy has been applied: 1) run a sequence of tasks 100000 times (without duplication parallelism) and 2) print the latency of the tasks. Then, knowing the sequential duration of each task, it is up to the system designer to find an efficient pipeline stages decomposition. Listing 7 presents the C++ DSEL source code corresponding to the pipeline description in Fig. 7a. Each task t_i is contained (as a method) in the M_i module (or class). The four main steps are: 1) Creation of the modules; 2) Binding of the tasks; 3) Creation of the pipeline strategy; 4) Pipeline execution.

Fig. 7b presents the internal structure of the pipeline. As we can see, new tasks have been automatically added: $push_{1\to n}$, $pull_{1\to n}$ shared by a *1 to n adaptor* module and $push_{n\to 1}$, $pull_{n\to 1}$ shared by a *n to 1 adaptor* module. The binding has been modified to insert the tasks of the adaptors. In the initial pipeline description, t_1 is bound to t_2. In a parallel pipelined execution this is not possible anymore because many threads are running concurrently: One for stage 1, four for stage 2 and one for stage 3 in the example. To this purpose, the adaptors implement a producer-consumer scheme. The yellow diamonds represent the buffers that are required. The $push_{1\to n}$ and $pull_{n\to 1}$ tasks can only be executed by a single thread while the $pull_{1\to n}$ and $push_{n\to 1}$ tasks are thread-safe. The $push_{1\to n}$ task copies its input socket in one buffer each time it is called. There is one buffer per duplicated sequence. To guarantee that the order of the input frames is preserved, a round-robin scheduling has been adopted. On the other side, the $pull_{n\to 1}$ task is copying the data from the buffers to its output socket, with the same round-robin scheduling.

The size of the synchronization buffers in the adaptors are defined on creation. The default size is one. During the copy of the input socket data in one of the buffers, threads cannot access the data until the copy is finished. The synchronization is automatically managed by the framework. If the buffer is full, the producer ($push_{1\to n}$ and $push_{n\to 1}$ tasks) has to wait. The same applies for the consumer ($pull_{1\to n}$ and $pull_{n\to 1}$ tasks) if the buffer is empty. We implemented both active and passive waiting.

Copies from and to buffers are expensive. These copies are removed by dynamically re-binding the tasks just before and just after the *push* and *pull* tasks, and casting the tasks into copyless variants. It is also necessary to bypass the regular execution in the $push_{1\to n}$, $pull_{1\to n}$, $push_{n\to 1}$ and $pull_{n\to 1}$ tasks. This replaces the source code of the data buffer copy by a simple pointer copy. The pointers are exchanged cyclically.

In Fig. 7b, the pipeline threads are pinned to specific CPU cores. This is the direct consequence of the lines 13-15 in Listing 7. The *hwloc* library[59] has been used and integrated in our DSEL to pin the software threads to processing units (hardware threads). In the given example, we assume that the CPU cores can only execute one hardware thread (SMT off). The threads pinning is given by the designer. This can improve the multi-threading performance on NUMA architectures.

Fig. 7 is an example of a simple chain of tasks. More complicated task graphs can have more than two tasks to synchronize between two pipeline stages. The adaptor implementation can manage multiple socket synchronizations. The key idea is to deal with a 2-dimensional array of buffers. Another difficult case is when a task t_1 is in stage 1 and possesses an output socket bound to an other task t_x which is located in the stage 4. To work, the pipeline adaptors between the stages 1 and 2 and the stages 2 and 3 automatically synchronize the data of the t_1 output socket.

5 | APPLICATION ON THE DVB-S2 STANDARD

In this section, we present a real use case of our DSEL. The second generation of Digital Video Broadcasting standard for Satellite (DVB-S2)[60] 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 where

[‡]USRP N320: https://www.ettus.com/all-products/usrp-n320/.

all the digital processing of the system have been implemented. The purpose of this section is not to detail all the implemented tasks extensively, but rather to expose the system as a whole. Some specific focuses are given to describe the main encountered problems and solutions.

5.1 | Transmitter Software Implementation

TABLE 2 Selected DVB-S2 configurations (MODCOD).

Config.	Modulation	Rate R	K_{BCH}	K_{LDPC}	N_{LDPC}	N_{PLH}	Interleaver
MODCOD 1	QPSK	3/5	9552	9720	16200	16740	no
MODCOD 2	QPSK	8/9	14232	14400	16200	16740	no
MODCOD 3	8-PSK	8/9	14232	14400	16200	16740	column/row

The DVB-S2 coding scheme rests upon the serial concatenation of a Bose, Ray-Chaudhuri & Hocquenghem (BCH) and a LDPC code. The selected modulation is a Phase-Shift Keying (PSK). The standard defines 32 MODulation and CODing schemes or *MODCODs*. This work focuses on the 3 MODCODs given in Tab. 2. 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 uses a column/row interleaver. K_{BCH} or K is the number of information bits and the input size of the BCH encoder. N_{BCH} or K_{LDPC} is the output size of the BCH encoder and the input size of the LDPC encoder. For each selected MODCOD, $N_{\text{LDPC}} = 16200$. With the pay load header (PLH) and pilots bits, the frame size (N_{PLH} or N) contains a total of 16740 bits.

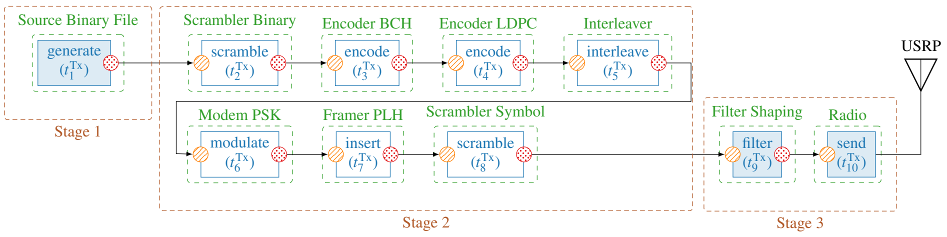

Source Binary File
generate (t_1^{Tx})
Stage 1
Scrambler Binary
scramble (t_2^{Tx})
Encoder BCH
encode (t_3^{Tx})
Encoder LDPC
encode (t_4^{Tx})
Interleaver
interleave (t_5^{Tx})
USRP
Modem PSK
modulate (t_6^{Tx})
Framer PLH
insert (t_7^{Tx})
Scrambler Symbol
scramble (t_8^{Tx})
Stage 2
Filter Shaping
filter (t_9^{Tx})
Radio
send (t_{10}^{Tx})
Stage 3

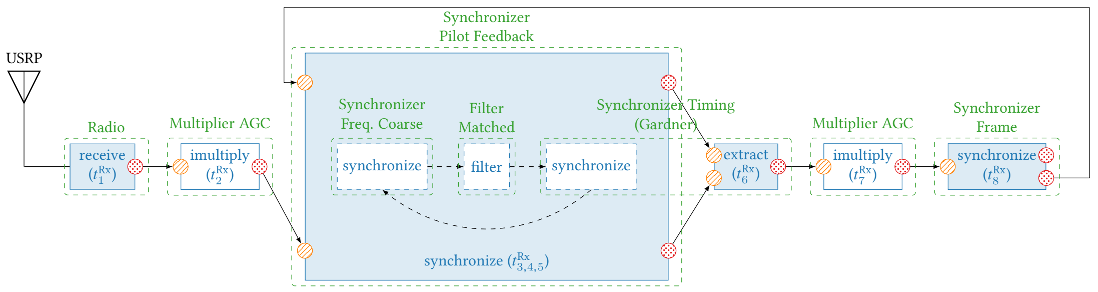
FIGURE 8 DVB-S2 transmitter software implementation.

Fig. 8 shows the DVB-S2 transmitter decomposition in tasks and pipeline stages. Intrinsically sequential tasks are represented by plain boxes. The DVB-S2 transmitter has been implemented in software with the proposed DSEL. Out of conciseness, it is not detailed in this paper as it is much more simpler than the receiver part of the system in terms of computational requirement and complexity of the task graph.

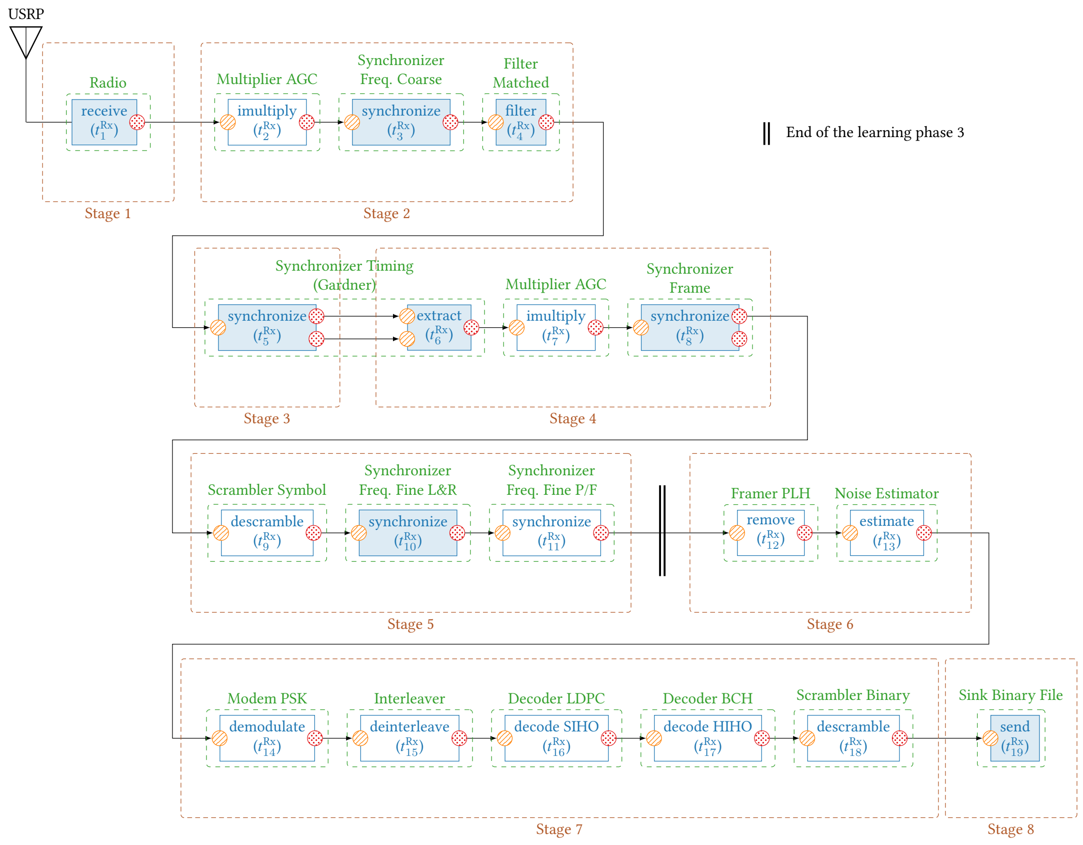
5.2 | Receiver Software Implementation

Fig. 9 presents the task decomposition of the DVB-S2 receiver software implementation with the five distinct phases. The plain tasks are intrinsically sequential and cannot be duplicated. The first phase is called the *waiting phase* (see Fig. 9a). 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 *learning phase 1* (see Fig. 9a) is executed during 150 frames. After that the *learning phase 2* (see Fig. 9a) is also executed during 150 frames. After the *learning phase 1 and 2*, the tasks have to be re-bound for the *learning phase 3* (see Fig. 9b). This last learning phase is applied over 200 frames. After the 500 frames of these successive learning phases, the final *transmission phase* is established (see Fig. 9b).

In a real life communication systems, the internal clocks of the radios 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 frequency



(a) Waiting phase and learning phase 1 & 2.



(b) Learning phase 3 & transmission phase.

FIGURE 9 DVB-S2 receiver software implementation.

does not perfectly match the receiver frequency, so the *Synchronizer Frequency* tasks (t_3^{Rx}, t_{10}^{Rx} and t_{11}^{Rx}) recalibrate the signal to recover the transmitted symbols. Finally LDPC decoder is a block coding scheme that requires to know precisely the first and last bits of the codeword. The *Synchronizer Frame* task (t_8^{Rx}) uses the PLH and pilots bits inserted by the transmitter to recover the first and last symbols. 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 output sockets, 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_5^{Rx}) to screen which data is selected for the next task. The *Synchronizer Timing* tasks (t_5^{Rx} and t_6^{Rx}) have an high latency compared to the others tasks, thus splitting the treatment in two tasks is a way to increase the throughput of the pipeline.

Besides in some cases the task does not have enough samples to produce a frame. In such cases, the *extract* task raises an abort exception (= early exit). The exception is caught and the sequence restarts from the first task (t_1^{Rx}).

During the waiting and learning phases 1 and 2, the *Synchronizer Freq. Coarse*, the *Filter Matched* and a part of the *Synchronizer Timing* have to work symbol by symbol. They have been grouped in the *Synchronizer Pilot Feedback* task ($t_{3,4,5}^{Rx}$). $t_{3,4,5}^{Rx}$ also requires a feedback input from the *Synchronizer Frame* task (t_8^{Rx}). This behavior is no longer necessary in subsequent phases, so the $t_{3,4,5}^{\text{Rx}}$ task has been split in t_3^{Rx}, t_4^{Rx} and t_5^{Rx}. Consequently, the feedback from the t_8^{Rx} second output socket is left unbound.

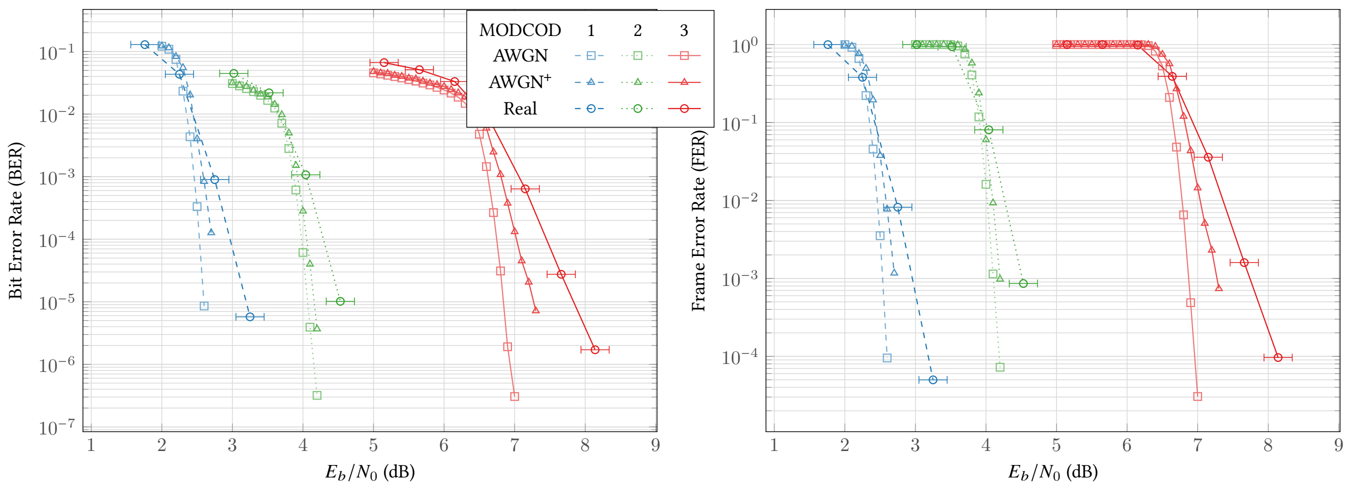


FIGURE 10 DVB-S2 BER & FER decoding performance (LDPC BP horizontal-layered, min-sum, 10 iterations).

Fig. 10 shows the frame error rate (FER) decoding performance results of the 3 selected MODCODs. The shapes represent the channel conditions: Squares stand for a standard simulated additive white Gaussian noise (AWGN) channel, triangles are 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 USRPs. There is 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. The MODCOD 1 is represented by dashed lines, MODCOD 2 by dotted lines and MODCOD 3 by solid lines. For each MODCOD, the LDPC decoder is based on the belief propagation algorithm with horizontal layered scheduling (10 iterations) and with the min-sum node update rules. Each DVB-S2 configuration has a well-separated SNR predilection zone.

5.3 | Open Source Integration with AFF3CT Toolbox

The proposed software implementation of the DVB-S2 digital transceiver is open source[§]. It is described with the help of the AFF3CT toolbox[61]. AFF3CT is a library dedicated to the digital communication systems and more specifically to the channel decoding algorithms. In this paper, we extend AFF3CT with the presented DSEL 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 (tasks $t_{\{1,3,4,5,6\}}^{\text{Tx}}$ in Fig. 8 and tasks $t_{\{14,15,16,17,19\}}^{\text{Rx}}$ in Fig. 9b) and are optimized for efficiency. For instance, knowing that the LDPC decoding is one of the most compute intensive task, an existing high performance SIMD implementation is used, based on the portable MIPP library[62]. Additional AFF3CT tasks have been implemented specifically for this project. These new tasks mainly address two areas: signal synchronizations and filters, and real-time communications. One may note that the proposed DSEL is also open source, it is integrated to the AFF3CT library as a dependency. Moreover, it can be used standalone: it defines sequences and parallelism between tasks that can be defined by another library. The DSEL is packaged in the repository named “AFF3CT-core”[¶].

5.4 | Evaluation

This section evaluates the receiver part of the system. The transmitter part as it is not the most compute intensive part and high throughputs are much more easier to reach. All the presented results have been obtained on two high-end NUMA machines. One

[§]DVB-S2 digital transceiver repository: https://github.com/aff3ct/dvbs2.
[¶]AFF3CT-core DSEL repository: https://github.com/aff3ct/aff3ct-core.

TABLE 3 Tasks sequential throughputs and latencies of the DVB-S2 receiver (transmission phase, 16288 frames, inter-frame level = 16, MODCOD 2, error-free SNR zone, x86 target) 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

is composed by two Intel® Xeon™ Platinum 2.70 Ghz 8168 CPUs, 24 cores 128 GB RAM (denoted as x86). *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® 2.00 GHz CN9975 v2.1 CPUs, 28 cores, 256 GB of RAM (denoted as ARM). Each core is powered by NEON SIMD ISA. In the proposed implementation, the data are represented by 32-bit floating-point numbers. Data parallelism level is thus 16 for AVX-512F ISA and 4 for NEON ISA. For both targets, the GNU C++ compiler version 9.3 has been used with the following flags: `-O3 -march=native`.

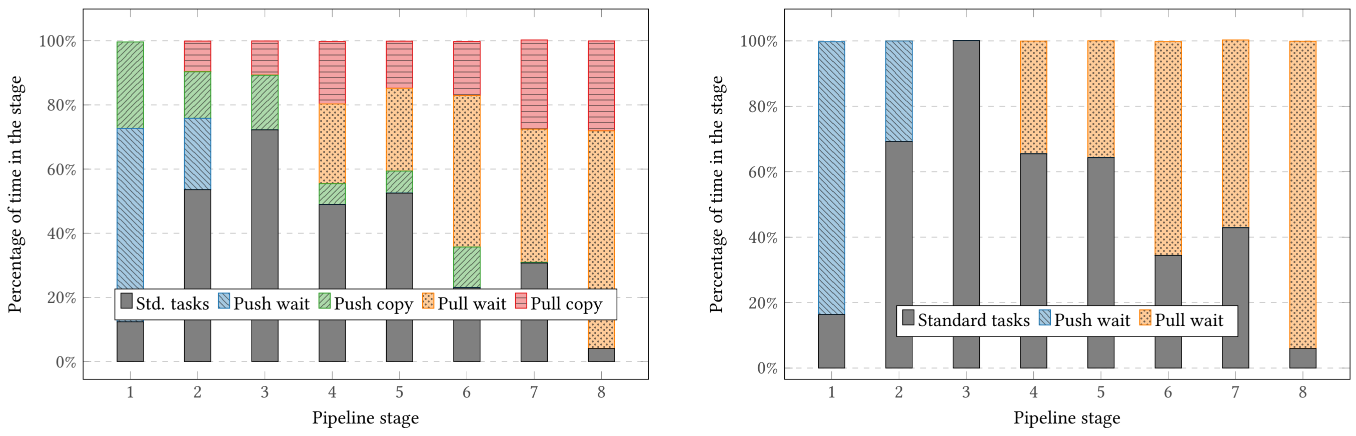
An high performance LDPC decoder implementation with the inter-frame SIMD technique is used (the early termination criterion has been switched on). This choice has the effect of computing sixteen/four frames at once in each task of the receiver (depending on the x86 or ARM target). It negatively affects the overall latency of the system (by a factor of sixteen/four). But it is not important in the video streaming targeted application. The *Decoder LDPC* task (t_{16}^{Rx}) is the only one in the receiver to take advantage of the inter-frame SIMD technique. The other tasks simply process sixteen/four frames sequentially.

Tab. 3 presents the tasks throughputs and latencies measured for a sequential execution of the MODCOD 2 in the transmission phase (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 ($K = 14232$). 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 many time compared to the other tasks. However, we chose to not spend too much time in optimizing the BCH decoding process as the stage 7 throughput can easily be increased with the sequence duplication technique. The second slower stage in 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. If a machine with an infinite number of cores is considered, the maximum reachable information throughput is 55.42 Mb/s.

We did not try to parallelize the waiting and the learning phases. We measured that the whole learning phase (1, 2 and 3) takes about one second. During the learning 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 learning 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 stages as presented in Fig. 9b. 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 an higher 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 the sequence duplication has been applied. The stage 7 has been parallelized 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. An early termination criterion was enabled. When the signal quality is very good, the *Decoder LDPC* task runs fast and the threads can spend a lot of time in waiting. With the passive waiting version of the adaptor *push* and *pull* tasks, the CPU dynamically adapt the cores charge and energy can be saved. In Tab. 3, 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* 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. The strategy of the stage 7 parallelism is to maximize the throughput. During the duplication process (modules clones), the thread pinning is known and the memory is 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 (adaptor pushes and pulls), 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 needs around 1.3 GB of the global memory during a sequential execution while it needs around 30 GB in parallel. The memory usage increases because of the sequence duplications in the stage 7. The duplication operation takes about 20 seconds. It is made at the very beginning of the program (before the waiting phase). It is worth mentioning that the amount of memory was not a critical resource. So, we did not try to reduce its overall occupancy.



(a) Data copy (stage throughput is 40 Mb/s). **(b)** Copy-less (stage throughput is 55 Mb/s).

FIGURE 11 Comparison of the two pipeline implementations in the receiver (x86 target, MODCOD 2).

Fig. 11 presents the repartition of the time in the pipeline stages on the x86 target (MODCOD 2). The receiver is running over 35 threads. Fig. 11a shows the pipeline implementation with data copies. Fig. 11b shows the pipeline implementation with pointer copies (copy-less). *Push wait* and *Pull wait* are the percentage of time spent in passive or active waiting. *Push copy* and *Pull copy* are the percentage of time spent in copying the data to and from the adaptors buffers. *Standard tasks* is the cumulative percentage of time spent by the tasks presented in Fig. 9b. In both implementations the pipeline stage throughput is constraint by the slowest one. In Fig. 11a the measured throughput per stage is 40 Mb/s whereas in Fig. 11b the measured throughput is 55 Mb/s. The copy-less implementation throughput is ≈ 27% higher than the data copy implementation. Fig. 11a shows that the copy overhead is non-negligible. A 27% slowdown is directly due to these copies in the stage 3. It largely justifies the copy-less implementation. In Fig. 11b and in the stage 3, 100% of time is taken by the t_5^{Rx} task. This is also confirmed by the measured throughput (55 Mb/s) which is very close the sequential throughput (55.42 Mb/s) reported in Tab. 3.

TABLE 4 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

Tab. 4 summarizes sequential and parallel throughputs for the 3 MODCODs presented in Tab. 2. To measure the maximum achievable throughput, the USRP modules are removed and samples are read from a binary file. 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 to be just a little bit slower than the slowest stage. Otherwise the radio task has to indefinitely buffer samples even though the amount of available memory in the machine is not infinite. The information throughput (K bits) is the final useful throughput for the user. Between the MODCOD 1 and 2, only the LDPC code rate varies ($R = 3/5$ and $R = 8/9$ resp.). In the parallel implementation, it has a direct impact on the information throughput. Between the MODCOD 2 and 3, the modulation varies (QPSK and 8-PSK resp.) and the frames have to be deinterleaved (column/row interleaver). High order modulation reduces the amount of samples processed in the *Synchronizer Timing* task (t_5^{Rx}): this results in higher throughput (80 Mb/s for the 8-PSK) in the slowest stage 3. In the parallel implementation, the pipeline stage throughputs are adapting to the slowest stage 3. It results in an important speedup. In the sequential implementation, it results in a little slowdown. Indeed, the additional time spent in the *deinterleave* task (t_{15}^{Rx}) is higher than the time saved in the *Synchronizer Timing* task (t_5^{Rx}).

These results demonstrate the benefit of our parallel implementation. Throughput speedups range from 10 to 20 compared to the sequential implementation. Selected configurations each are most efficient in different SNR zones (as shown in Fig. 10), depending on the signal quality. For instance, MODCOD 1 is adapted for noisy environments (3 dB). However the information throughput is limited to 37 Mb/s (x86 target). MODCOD 3 is more adapted to clearer signal conditions (7.5 dB) and the information throughput reaches 80 Mb/s (x86 target). MODCOD 2 is in-between. 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). However, being able to run the transceiver on both x86 and ARM CPUs with comparable throughput demonstrates the flexibility and the portability of the proposed framework.

5.5 | Comparison with State-of-the-Art

gr-dvbs2rx

To the best of our knowledge, it is the faster open source implementation at the time of the writing of the paper. gr-dvbs2rx has been run on the same x86 target presented before (with the same compiler and options) and on the MODCOD 2. We ran the code without the radios, this way only the software part of the receiver is evaluated. First, a set of IQs have been generated from the emitter and written on a file. Then, the receiver has been executed on it. The set of IQs have been read from the same file. The evaluation has been made on error-free SNR zone. To make a fair comparison, we modified a little bit the source code of the receiver to remove "artificial blocks" that slowed down the throughput. The throttle block has been removed as well as some useless (and not optimized) blocks dedicated to the conversion of the IQs (from 8-bit fixed-point to 32-bit floating-point format). The source code modifications we made are available on a fork of the project[#].

TABLE 5 gr-dvbs2rx throughputs per pipeline stage compared with this work (error-free SNR zone, x86 target, MODCOD 2). Same color codes as in Tab. 3.

GNU Radio		Equi.	Throughput (Mb/s)	
Stage	Block	$(t^{\mathrm{Rx}}_{\{i\}})$	gr-dvbs2rx	This work
1	*file_source*	1	100.7	431.8
2	*agc_cc*	2	24.0	367.5
3	*symbol_sync_cc*	4-6	16.9	33.1
4	*rotator_cc*	7	96.1	685.5
5	*plsync_cc*	3,8-13	45.3	48.4
6	*ldpc_decoder_cb*	14-16	23.0	31.7
7	*bch_decoder_bb*	17	14.9	6.9
8	*bbdescrambler_bb*	18	225.8	91.1
9	*bbdeheader_bb*	–	252.5	–
10	*file_sink*	19	346.5	1838.3

Tab. 5 presents the per block normalized throughputs of gr-dvbs2rx and of this work (considering the gr-dvbs2rx pipeline decomposition). For each GNU Radio block the tasks equivalence with our system is given. We measured an overall information throughput of 14.9 Mb/s. As GNU Radio pins each block to a thread, the throughput performance is driven by the slowest block (here *bch_decoder_bb*). gr-dvbs2rx uses 10 threads (same as the number of stages). If we applied the same decomposition without the duplication of the stage 7 our receiver will have been limited to the throughput of the BCH decoder (6.9 Mb/s). With the duplication technique the stage 7 is automatically dispatched on multiple threads and its throughput is approximately multiplied by the number of threads. This automatic transformation is not possible with the GNU Radio implementation. Moreover, still if we only consider the gr-dvbs2rx pipeline decomposition, our work will have been limited to the lowest sequential throughput which is 33.1 Mb/s (*symbol_sync_cc* block). Thanks to a finer decomposition into tasks ($symbol\_sync\_cc = t^{\mathrm{Rx}}_4 + t^{\mathrm{Rx}}_5 + t^{\mathrm{Rx}}_6$) our receiver is able to reach 55 Mb/s (see Tab. 4). As a result, the throughput of the proposed implementation is 3.7 times faster than gr-dvbs2rx.

Grayver and Utter

For a comparable CPU (Intel® Xeon™ E5-2698v4), and on a same code rate ($R = 1/2$), their solution reaches an information throughput of 180 Mb/s, this is 3.3 times higher than what is achieved with the proposed implementation (55 Mb/s). This is mainly due to new algorithmic improvements in the synchronization tasks. For instance, they were able to express more parallelism in the *Synchronizer Timing* task (t^{Rx}_5). However, we also tried some aggressive optimizations in this task but we never succeeded to measure the same level of FER decoding performance. It could be interesting to check for any penalty in terms of decoding performance that may occur and to combine their optimizations with our DSEL. Unlike our work, their work focuses on a single DVB-S2 configuration (8-PSK, $N = 64800$ and $R = 1/2$) and a single architecture (x86). Their implementation looks like a hard-coded solution for the DVB-S2 standard while our goal is to provide generic methods and tools for SDR system implementations.

[#]gr-dvbs2rx fork (thr_benchmark branch): github.com/kouchy/gr-dvbs2rx/

6 | CONCLUSION

In this article, we introduced a new DSEL designed to satisfy SDR needs in terms of expressiveness and performance on multicore processors. It allows the definition of stateless/stateful tasks, dynamic control and early exits. First, we evaluated it on real-world representative micro-benchmarks and showed that its scheduling overhead is negligible for tasks longer than 4 μs. Then, we evaluated a full software implementation of the DVB-S2 standard built with our DSEL and the AFF3CT library for tasks. This implementation is the fastest open source software solution on multicore CPUs. It matches satellite real time constraints (30 - 50 Mb/s), which demonstrates the relevance and efficiency of the DSEL. This is the consequence of two main factors: 1) the low overhead achieved by DSEL, 2) an efficient implementation of the pipeline technique, where one stage, parallelized, reaches saturation. In future works, we plan to integrate the parallel features of the DSEL with high level languages such as Python or MATLAB® typically used in the signal processing community, often less familiar with the C++ language. Moreover, automatic parallelization, tuning of pipelining stages, could be investigated. A profile-guided optimization, capturing task runtime, has been used to tune pipeline stages, for instance. A more automatic and integrated approach could be possible since the analysis of the task graph is dynamic. Finally, the correctness of parallel stateful dataflow programs could be explored to guide the system designers in the development process.

References

1. Palkovic M, Raghavan P, Li M, Dejonghe A, Van der Perre L, Catthoor F. Future Software-Defined Radio Platforms and Mapping Flows. *IEEE Signal Processing Magazine* 2010; 27(2): 22–33. doi: 10.1109/MSP.2009.935386

2. Palkovic M, Declerck J, Avasare P, et al. DART - a High Level Software-Defined Radio Platform Model for Developing the Run-Time Controller. *Springer Journal of Signal Processing Systems (JSPS)* 2012; 69: 317–327. doi: 10.1007/s11265-012-0669-3

3. ETSI . 3GPP - TS 38.212 - Multiplexing and Channel Coding (R. 15). https://www.etsi.org/deliver/etsi_ts/138200_138299/138212/15.02.00_60/ts_138212v150200p.pdf; 2018.

4. Rost P, Bernardos CJ, Domenico AD, et al. Cloud Technologies for Flexible 5G Radio Access Networks. *IEEE Communications Magazine* 2014; 52(5): 68–76. doi: 10.1109/MCOM.2014.6898939

5. Mitola J. Software Radios: Survey, Critical Evaluation and Future Directions. *IEEE Aerospace and Electronic Systems Magazine* 1993; 8(4): 25–36. doi: 10.1109/62.210638

6. Akeela R, Dezfouli B. Software-Defined Radios: Architecture, State-of-the-Art, and Challenges. *ACM Computer Communications* 2018; 128: 106–125. doi: 10.1016/j.comcom.2018.07.012

7. Coulton P, Carline D. An SDR Inspired Design for the FPGA Implementation of 802.11a Baseband System. In: International Symposium on Consumer Electronics (ISCE). IEEE; 2004: 470–475

8. Skey K, Bradley J, Wagner K. A Reuse Approach for FPGA-Based SDR Waveforms. In: Military Communications Conference (MILCOM). IEEE; 2006: 1–7

9. Dutta P, Kuo Y, Ledeczi A, Schmid T, Volgyesi P. Putting the Software Radio on a Low-calorie Diet. In: Workshop on Hot Topics in Networks (HotNets). ACM; 2010

10. Shaik S, Angadi S. Architecture and Component Selection for SDR Applications. *International Journal of Engineering Trends and Technology (IJETT)* 2013; 4(4): 691–694.

11. Maheshwarappa MR, Bowyer M, Bridges CP. Software Defined Radio (SDR) Architecture to Support Multi-satellite Communications. In: Aerospace Conference (AeroConf). IEEE; 2015: 1–10

12. Nivin R, Rani JS, Vidhya P. Design and Hardware Implementation of Reconfigurable Nano Satellite Communication System using FPGA based SDR for FM/FSK Demodulation and BPSK Modulation. In: International Conference on Communication Systems and Networks (ComNet). IEEE; 2016: 1–6

13. Kaur G, Raj V. Multirate Digital Signal Processing for Software Defined Radio (SDR) Technology. In: International Conference on Emerging Trends in Engineering and Technology (ICETET). IEEE; 2008: 110–115

14. Karlsson A, Sohl J, Wang J, Liu D. ePUMA: A Unique Memory Access based Parallel DSP Processor for SDR and CR. In: Global Conference on Signal and Information Processing (GlobalSIP). IEEE; 2013: 1234–1237

15. Yoge DRN, Chandrachoodan N. GPU Implementation of a Programmable Turbo Decoder for Software Defined Radio Applications. In: International Conference on VLSI Design. IEEE; 2012: 149–154

16. Bang S, Ahn C, Jin Y, Choi S, Glossner J, Ahn S. Implementation of LTE System on an SDR Platform using CUDA and UHD. *Springer Journal of Analog Integrated Circuits and Signal Processing (AICSP)* 2014; 78(3): 599. doi: 10.1007/s10470-013-0229-1

17. Meshram S, Kolhare N. The Advent Software Defined Radio: FM Receiver with RTL SDR and GNU Radio. In: International Conference on Smart Systems and Inventive Technology (ICSSIT). IEEE; 2019: 230–235

18. Grayver E, Utter A. Extreme Software Defined Radio – GHz in Real Time. In: Aerospace Conference (AeroConf). IEEE; 2020.

19. Xianjun J, Canfeng C, Jääskeläinen P, Guzma V, Berg H. A 122Mb/s Turbo decoder using a mid-range GPU. In: International Wireless Communications and Mobile Computing Conference (IWCMC). IEEE; 2013: 1090–1094

20. Li R, Dou Y, Xu J, Niu X, Ni S. An Efficient Parallel SOVA-based Turbo Decoder for Software Defined Radio on GPU. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 2014; 97(5): 1027–1036. doi: 10.1587/transfun.E97.A.1027

21. Le Gal B, Jégo C, Crenne J. A High Throughput Efficient Approach for Decoding LDPC Codes onto GPU Devices. *IEEE Embedded Systems Letters (ESL)* 2014; 6(2): 29–32. doi: 10.1109/LES.2014.2311317

22. Giard P, Sarkis G, Leroux C, Thibeault C, Gross WJ. Low-Latency Software Polar Decoders. *Springer Journal of Signal Processing Systems (JSPS)* 2016; 90: 761–775. doi: 10.1007/s11265-016-1157-y

23. Keskin S, Kocak T. GPU Accelerated Gigabit Level BCH and LDPC Concatenated Coding System. In: High Performance Extreme Computing Conference (HPEC). IEEE; 2017: 1–4

24. Sarkis G, Giard P, Thibeault C, Gross WJ. Autogenerating Software Polar Decoders. In: Global Conference on Signal and Information Processing (GlobalSIP). IEEE; 2014: 6–10

25. Le Gal B, Leroux C, Jégo C. Multi-Gb/s Software Decoding of Polar Codes. *IEEE Transactions on Signal Processing (TSP)* 2015; 63(2): 349–359. doi: 10.1109/TSP.2014.2371781

26. Cassagne A, Le Gal B, Leroux C, Aumage O, Barthou D. An Efficient, Portable and Generic Library for Successive Cancellation Decoding of Polar Codes. In: International Workshop on Languages and Compilers for Parallel Computing (LCPC). Springer; 2015

27. Sarkis G, Giard P, Vardy A, Thibeault C, Gross WJ. Fast List Decoders for Polar Codes. *IEEE Journal on Selected Areas in Communications (JSAC)* 2016; 34(2): 318–328. doi: 10.1109/JSAC.2015.2504299

28. Cassagne A, Tonnellier T, Leroux C, Le Gal B, Aumage O, Barthou D. Beyond Gbps Turbo decoder on multi-core CPUs. In: International Symposium on Turbo Codes and Iterative Information Processing (ISTC). IEEE; 2016: 136–140

29. Cassagne A, Aumage O, Leroux C, Barthou D, Le Gal B. Energy Consumption Analysis of Software Polar Decoders on Low Power Processors. In: European Signal Processing Conference (EUSIPCO). IEEE; 2016: 642–646

30. Le Gal B, Jégo C. High-Throughput Multi-Core LDPC Decoders Based on x86 Processor. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 2016; 27(5): 1373–1386. doi: 10.1109/TPDS.2015.2435787

31. Le Gal B, Jégo C. Low-Latency Software LDPC Decoders for x86 Multi-Core Devices. In: International Workshop on Signal Processing Systems (SiPS). IEEE; 2017: 1–6

32. Léonardon M, Cassagne A, Leroux C, Jégo C, Hamelin LP, Savaria Y. Fast and Flexible Software Polar List Decoders. *Springer Journal of Signal Processing Systems (JSPS)* 2019; 91: 937–952. doi: 10.1007/s11265-018-1430-3

33. Le Gal B, Jégo C. Low-latency and High-throughput Software Turbo Decoders on Multi-core Architectures. *Springer Annals of Telecommunications* 2019; 75: 27–42. doi: 10.1007/s12243-019-00727-5

34. Dennis J. Data Flow Supercomputers. *IEEE Computer* 1980; 13(11): 48–56. doi: 10.1109/MC.1980.1653418

35. Ackerman W. Data Flow Languages. *IEEE Computer* 1982; 15: 15–25. doi: 10.1109/MC.1982.1653938

36. Lee EA, Messerschmitt DG. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers (TC)* 1987; C-36(1): 24–35. doi: 10.1109/TC.1987.5009446

37. Engels M, Bilsen G, Lauwereins R, Peperstraete JA. Cycle-Static Dataflow: Model and Implementation. In: . 1. Asilomar Conference on Signals, Systems, and Computers (ACSSC). IEEE; 1994: 503–507

38. Bilsen G, Engels M, Lauwereins R, Peperstraete JA. Cyclo-Static Data Flow. In: . 5. International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE; 1995: 3255–3258

39. Parks TM, Pino JL, Lee EA. A Comparison of Synchronous and Cycle-Static Dataflow. In: . 1. Asilomar Conference on Signals, Systems, and Computers (ACSSC). IEEE; 1995: 204–210

40. Cassagne A, Léonardon M, Tajan R, et al. A Flexible and Portable Real-time DVB-S2 Transceiver using Multicore and SIMD CPUs. In: International Symposium on Topics in Coding (ISTC). IEEE; 2021

41. Thies W, Karczmarek M, Amarasinghe S. StreamIt: A Language for Streaming Applications. In: Horspool RN. , ed. *International Conference on Compiler Construction (CC)*Springer; 2002; Berlin, Heidelberg: 179–196

42. Buck I, Foley T, Horn D, et al. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics (TOG)* 2004; 23(3): 777–786. doi: 10.1145/1015706.1015800

43. Amarasinghe S, Gordon lM, Karczmarek M, et al. Language and Compiler Design for Streaming Applications. *Springer International Journal of Parallel Programming (IJPP)* 2005; 2(33): 261–278. doi: 10.1007/s10766-005-3590-6

44. Liao SW, Du Z, Wu G, Lueh GY. Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In: International Symposium on Code Generation and Optimization (CGO). IEEE; 2006: 207–219

45. Black-Schaffer D, Dally WJ. Block-Parallel Programming for Real-Time Embedded Applications. In: International Conference on Parallel Processing (ICPP). IEEE; 2010: 297–306

46. Glitia C, Dumont P, Boulet P. Array-OL with Delays, a Domain Specific Specification Language for Multidimensional Intensive Signal Processing. *Springer Multidimensional Systems and Signal Processing* 2010(21): 105–131. doi: 10.1007/s11045-009-0085-4

47. Thies W, Amarasinghe S. An Empirical Characterization of Stream Programs and its Implications for Language and Compiler Design. In: International Conference on Parallel Architectures and Compilation Techniques (PACT). ACM/IEEE; 2010: 365–376

48. De Oliveira Castro P, Louise S, Barthou D. DSL Stream Programming on Multicore Architectures. In: John Wiley and Sons. 2017

49. Dardaillon M, Marquet K, Risset T, Martin J, Charles HP. A New Compilation Flow for Software-Defined Radio Applications on Heterogeneous MPSoCs. *ACM Transactions on Architecture and Code Optimization (TACO)* 2016; 13(2). doi: 10.1145/2910583

50. Tan K, Liu H, Zhang J, Zhang Y, Fang J, Voelker GM. Sora: High-Performance Software Radio Using General-Purpose Multi-Core Processors. *ACM Communications* 2011; 54(1): 99–107. doi: 10.1145/1866739.1866760

51. Li R, Dou Y, Zhou J, Deng L, Wang S. CuSora: Real-time Software Radio using Multi-core Graphics Processing Unit. *Elsevier Journal of Systems Architecture (JSA)* 2014; 60(3): 280-292. doi: 10.1016/j.sysarc.2013.10.009

52. Hussain T, Khan M, Rehman MU, et al. A High Performance Software Defined Radio System Architecture and Development Environment for a Wide Range of Applications. In: International Conference on Computing, Mathematics and Engineering Technologies (iCoMET). IEEE; 2018: 1-5

53. Rondeau T, Blum J, Corgan J, et al. GNURadio: the Free and Open Software Radio Ecosystem. https://github.com/gnuradio/gnuradio; 2006.

54. Bloessl B, Müller M, Hollick M. Benchmarking and Profiling the GNU Radio Scheduler. In: . 4. GNU Radio Conference (GRCon). ; 2019.

55. Müller M. How to evolve the GNU Radio scheduler - Embracing and breaking legacy. In: FOSDEM. ; 2020.

56. pabr . leansdr: Lightweight, Portable Software-defined Radio.. https://github.com/pabr/leansdr; 2016.

57. Ryan W, Lin S. *Channel Codes: Classical and Modern*. Cambridge University Press . 2009

58. Freire I, Economos R, Inan A. gr-dvbs2rx: GNU Radio Extensions for the DVB-S2 and DVB-T2 Standards. https://github.com/igorauad/gr-dvbs2rx; 2022.

59. Broquedis F, Clet-Ortega J, Moreaud S, et al. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In: Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP). IEEE; 2010: 180–186

60. ETSI . EN 302 307 - Digital Video Broadcasting (DVB); Second Generation Framing Structure, Channel Coding and Modulation Systems for Broadcasting, Interactive Services, News Gathering and Other Broadband Satellite Applications (DVB-S2). https://www.etsi.org/deliver/etsi_en/302300_302399/302307/01.02.01_60/en_302307v010201p.pdf; 2005.

61. Cassagne A, Hartmann O, Léonardon M, et al. AFF3CT: A Fast Forward Error Correction Toolbox!. *Elsevier SoftwareX* 2019; 10: 100345. doi: 10.1016/j.softx.2019.100345

62. Cassagne A, Aumage O, Barthou D, Leroux C, Jégo C. MIPP: A Portable C++ SIMD Wrapper and its use for Error Correction Coding in 5G Standard. In: Workshop on Programming Models for SIMD/Vector Processing (WPMVP). ACM; 2018; Vösendorf/Wien, Austria

□