OTAC: Optimal Scheduling for Pipelined and Replicated Task Chains for Software-Defined Radio

Diane Orhan, Laércio Lima Pilla, Denis Barthou, Adrien Cassagne, Olivier Aumage, Romain Tajan, Christophe Jégo, Camille Leroux

OTAC: Optimal Scheduling for Pipelined and Replicated Task Chains for Software-Defined Radio

Diane Orhan[1], Laércio Lima Pilla[1], Denis Barthou[1], Adrien Cassagne[2],
Olivier Aumage[1], Romain Tajan[3], Christophe Jego[3], Camille Leroux[3]
[1] *Univ. Bordeaux, CNRS, Bordeaux INP, Inria, LaBRI, UMR 5800, Talence, France*
[2] *LIP6, Sorbonne Université, CNRS, Paris, France*
[3] *Univ. Bordeaux, CNRS, Bordeaux INP, IMS, UMR 5218, Talence, France*
diane.orhan@inria.fr

***Abstract*—Software-Defined Radio (SDR) represents a move from dedicated hardware to software implementations of digital communication standards. This approach offers flexibility, shorter time to market, maintainability, and lower costs, but it requires an optimized distribution of SDR tasks in order to meet performance requirements. In this context, we study the problem of scheduling SDR linear task chains of stateless and stateful tasks. We model this problem as a pipelined workflow scheduling problem based on pipelined and replicated parallelism on homogeneous resources. Based on this model, we propose a scheduling algorithm named OTAC for maximizing throughput while also minimizing the number of allocated hardware resources, and we prove its optimality. We evaluate our approach and compare it to other algorithms in a simulation campaign, and with an actual implementation of the DVB-S2 communication standard on the AFF3CT SDR Domain Specific Language. Our results demonstrate how OTAC finds optimal schedules, leading consistently to better results than other algorithms, or equivalent results with much fewer hardware resources.**

***Index Terms*—software-defined radio, pipelined worfklow scheduling, pipelining, replication, optimal algorithm, task chains**

I. Introduction

Digital communication standards have traditionally been implemented on dedicated hardware in order to meet real-time constraints. These implementations achieve high performance and low energy consumption at the expense of flexibility and high development costs [1]. Such issues have been increasing in importance as communications standards, such as the 5G mobile standard, evolved to handle usage profiles in a versatile manner [2], fueling a transition from hardware to software implementations.

Software implementations offer flexibility, maintainability, shorter time to market, and lower costs (e.g., by sharing hardware). This approach is seen in the notion of **Software-Defined Radio (SDR)**, where components of a digital communication chain responsible for the physical layer (layer 1 in the OSI model) and the media access control sublayer (layer 2) are implemented in software and executed on traditional processors or reconfigurable hardware [3]–[8]. A challenge in this scenario lies in distributing these computations over the available hardware while meeting performance requirements.

The computations of digital communications are implemented as a series of processing blocks, called *tasks*, connected in a directed graph (linear *chains*, in our context). These tasks can be grouped and partitioned in a sequence of stages in a **pipeline**, where each stage is mapped to a different core, to improve throughput. Nonetheless, pipelining alone may not be sufficient to maximally use the available hardware parallelism.This limitation can be overcome by **replicating** some pipeline stages on different cores and distributing frames in a round-robin fashion among replicas [9]. Still, replication is limited to stages composed of *stateless*[1] tasks only, to ensure consistency with the original sequential task execution.

The combination of pipelining and replication provides a framework to achieve high throughput when scheduling tasks [7], but also economic and (indirect) environmental benefits. For instance, by making the best use of the available multicore hardware on antennas, one may avoid or delay the expenses and emissions related to acquiring new specific computing hardware to handle new digital communication standards. However, **finding the best schedule is challenging**. Automatic approaches have so far been limited to: Applying only pipelining [10], [11] or replication [12]; Using Integer Linear Programming [9], [13], [14], with potential prohibitive computation time; Using heuristics [14], [15] and trading optimality for a faster computation time; Or avoiding these decisions altogether by assigning a single task per thread and leaving the scheduling work to the operating system [16].

We propose an optimal algorithm, named OTAC, for scheduling pipelined and replicated task chains on homogeneous computing resources for SDR. The schedule first maximizes throughput and then minimizes the number of required cores, based on the characteristics of the communication chain and a given maximum number of cores. Our main contributions are the following:

- ⋆ We provide a formulation of this throughput optimization problem (Section III).
- ⋆ We propose a new pseudo-polynomial time algorithm named OTAC, and we prove its optimality (Section IV).
- ⋆ We evaluate its schedule using simulated task chains and an implementation of the Digital Video Broadcasting – Second Generation (DVB-S2) standard [17] (Section V).

[1] *Stateless* tasks depend only on their current input (e.g., a filter), while *stateful* tasks are also influenced by their internal state (due to previous inputs).

The remaining sections of this article are organized as follows: Section II provides an overview of related work, and Section VI provides concluding remarks.

II. Related Work

We break the discussion into two parts: pipelined workflow scheduling problems, and solutions for SDR.

A. Pipelined workflow scheduling problems

Our scheduling problem is related to **pipelined workflow scheduling** problems focused on throughput optimization. A survey by Benoit *et al.* [18] distinguishes four kinds of parallelism in these problems: (i) *Task parallelism*, two or more independent tasks execute simultaneously on the same data; (ii) *Pipelined parallelism*, two dependent tasks execute on different data; (iii) *Data parallelism*, multiple resources execute the same task on parts of the same data; iv *Replicated parallelism*, several copies of a single task operate on different data. We focus only on pipelined and replicated parallelism.

When *only pipelining is possible (all tasks are stateful)*, the problem becomes a classical chains-to-chains partitioning (CCP) problem with known optimal solutions [10], [11], [19], [20]. For instance, Nicol's algorithm [11], [20] optimally minimizes the duration of the longest pipeline stage by means of a dichotomic search on the possible maximal stage duration and of a greedy approach to partition the tasks among the pipeline stages. Meanwhile, *when replication is possible and all tasks are stateless*, an optimal solution can be found by creating a single partition with all tasks and replicating it among all available resources [12]. However, *no optimal solution has been proposed so far using pipelining and replication when both stateless and stateful tasks are present.*

In the related (but different) context of latency minimization, it is known that pipelining and replication have no positive effect on latency for task chains [12], [18]. For more generic directed acyclic graphs (DAGs), the problem is known to be NP-Complete. Kwok and Ahmad [21] provide an extensive survey on the topic. Finally, heuristics proposed for related objectives and DAGs are usually not applicable in our context. For instance, Vydyanathan *et al.* [22] propose a heuristic for optimizing both latency and resources used under throughput constraints using pipelining and replication. Nonetheless, the solution assumes that all tasks can be replicated, which is not possible in our context because stateful tasks are not replicable.

B. Solutions employed on SDR or signal processing systems

There are multiple frameworks and tools dedicated to the design and implementation of SDR or signal processing systems. Each framework implements and schedules tasks in its own way, which has led to different optimization solutions.

GNU Radio [23] is the most widely used open source framework in this domain. It provides an extensible library of signal processing *blocks* (tasks) to be combined in a flowgraph. In its current implementation (version 3.0), GNU Radio employs Thread Per Block scheduling [16]. Thus, each task is assigned to its own thread, and thread management is left to the operating system (or to another runtime such as CEDR [24] for exploring heterogeneous architectures). Although simpler, this approach can lead to performance problems related to locality, to thread priorities and scheduling policies [16], or even related to the overhead of managing too many threads per core [25]. For these reasons, future plans for GNU Radio 4.0 include the removal of this constraint, allowing more than one block per thread and better scheduling policies [25]. In this sense, we believe GNU Radio can benefit from OTAC in the future.

StreamIt is a Domain Specific Language (DSL) that provides an environment for developing SDR chains [26]. Unlike GNU Radio, StreamIt has its own scheduling algorithm built into its compiler [27]. It is based on a greedy algorithm that applies pipelined parallelism (*vertical fusion* in its context) when fewer cores than tasks are available, or replicated parallelism (*horizontal fission*) when cores are abundant [27]. Meanwhile, OTAC uses both kinds of parallelism to maximize throughput while using the least number of cores possible.

Other approaches have been proposed for StreamIt. Kudlur and Mahlke [9] use Integer Linear Programming (ILP) to model the problem of scheduling StreamIt stream graphs. As ILP is NP-Hard and the model makes individual decisions for each task, this approach is not considered scalable [15]. Yi *et al.* [13] proposed grouping non-contiguous tasks together into batches that are then scheduled using ILP. This approach trades optimality for faster decisions. Che, Panda, and Chatha [14] apply a similar ILP formulation but also propose a heuristic that fuses stateful batches to free resources for replicating the slowest batches. Gayen *et al.* [15] avoid batching tasks by relaxing the ILP formulation into a Mixed Integer Linear Programming format and combining it with a rounding heuristic in order to accelerate decisions. These approaches have the drawback of trying to assign all available resources, which can lead to performance losses due to overhead [15]. Additionally, by bunching together non-contiguous tasks, communication overhead is increased by having to transfer data extraneous times between cores. In contrast, OTAC provides optimal schedules and fast decisions for digital communication chains while also avoiding additional communication overhead and resorting to the minimal amount of resources.

Array-OL is a DSL for processing multidimensional streams for video streaming [28]. Similarly to StreamIt [27], it leaves the scheduling optimization process to its compiler. Nonetheless, one of its main contributions comes from exploring the *data parallelism* in the multidimensional streams. This kind of parallelism is not considered in this work.

Finally, **AFF3CT** is a DSL that provides pipelined and replicated parallelism for SDR [7], [29]. When a contiguous sequence of tasks is assigned to the same pipeline stage, AFF3CT's precompiler is able to fuse these tasks to avoid unnecessary data transfers and synchronization. Nonetheless, choosing which tasks go in the same pipeline stage and which stages are replicated is left to the designer [30]. We overcome this issue by proposing an automatic and optimal schedule.

III. Problem Definition

The optimization of SDR communication chains' throughput can be depicted as a pipelined workflow scheduling problem with special constraints. These problems can be characterized by their workflow model, system model, performance model, and mapping strategy [18].

The workflow can be described as a linear chain of N tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_N\}$, meaning τ_i can only execute after τ_{i-1}. A task τ_i has a computation weight w_i (i.e., its latency). Communication weights are not explicitly considered. All tasks are sequential. Regarding their execution, tasks are partitioned into two subsets $\mathcal{T}_F$ and $\mathcal{T}_L$ for *stateful* and *stateless* tasks.

Our system is composed of P homogeneous resources (e.g., cores) that are fully-connected. Other network characteristics do not play a role in this problem. Regarding performance, our main objective is to *maximize throughput* (T) while also minimizing the number of resources assigned P'. As throughput is related to the stages' weight, the notation T^{-1} is used to represent the *reciprocal throughput to be minimized*.

The mapping strategy of our problem is known as interval mapping and can be defined as a vector composition $(\mathbf{n}, \mathbf{p})$ of (N, P): $\mathbf{n}$ and $\mathbf{p}$ are finite suites $\mathbf{n} = (n_1, \ldots, n_k)$ and $\mathbf{p} = (p_1, \ldots, p_k)$ verifying $\sum_{i=1}^{k} n_i = N$ and $\sum_{i=1}^{k} p_i \leq P$. $\mathbf{n}$ defines sub-sequences of n_i consecutive tasks and $\mathbf{p}$ defines the number p_i of resources dedicated to a sub-sequence. We use s_i to represent the subset of tasks in sub-sequence n_i. For example, if $n_1 = 3$ and $n_2 = 2$, then $s_1 = \{\tau_1, \tau_2, \tau_3\}$ and $s_2 = \{\tau_4, \tau_5\}$. We call the pair (s_i, p_i) a *stage*. s_i is defined as stateless if all its tasks are stateless, otherwise it is stateful. Stateful sub-sequences can only benefit from one resource.

We define the successor of a task τ_i as the task τ_{i+1} $(1 \leq i < N)$. The successor of a sub-sequence s_i is the task immediately following the last task of s_i, if any.

The weight of a stage is defined by the function w (Eq. (1)). It is equal to the sum of the weights of its tasks. This value is divided by the number of resources on stateless stages. The reciprocal throughput T^{-1} of the vector composition is the greatest weight among all stages (Eq. (2)). Our objective is to find a composition of (N, P) minimizing T^{-1} and P'.

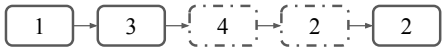
$$w(s_i, p_i) = \begin{cases} \sum_{\tau \in s_i} w_\tau & \text{if } s_i \cap \mathcal{T}_F \neq \emptyset,\ p_i \geq 1, \\ \frac{1}{p_i} \sum_{\tau \in s_i} w_\tau & \text{if } s_i \cap \mathcal{T}_F = \emptyset,\ p_i \geq 1, \\ \infty & \text{otherwise} \end{cases} \quad (1)$$

$$T^{-1} = T(\mathbf{n}, \mathbf{p}) = \max_{i=1,\ldots,k} w(s_i, p_i) \quad (2)$$

The optimal solutions to two extreme cases are known (see Section II-A). If all tasks are stateful, the problem becomes a classical CCP problem [11]. If all tasks are stateless, it is optimal to have a single partition using all resources [12].

A. Communication Chain and Scheduling Example

Consider the simple case of the SDR digital communication standard (chain) in Fig. 1a composed of $N = 5$ tasks with computations weights $w_1 = 1$, $w_2 = 3$, $w_3 = 4$, $w_4 = w_5 = 2$. Tasks τ_1, τ_2, and τ_5 are stateful, while τ_3 and τ_4 are stateless (i.e., $\mathcal{T}_F = \{\tau_1, \tau_2, \tau_5\}$ and $\mathcal{T}_L = \{\tau_3, \tau_4\}$). Its optimal schedule is studied in two situations with 3 or 6 resources.



(a) Chain composed of 5 tasks. Each box represents a task. The number inside represents its weight. Boxes with solid lines represent stateful tasks, while dashed lines represent stateless tasks.

(b) Optimal schedule for $P = 3$. Vertical lines represent the separation in stages.

(c) Optimal schedule for $P = 6$. Tasks in s_3 are duplicated to represent $p_3 = 2$. Only $P' = 5$ is necessary for the optimal solution.

Fig. 1: Example of a task chain and its optimal schedule for 3 and 6 resources.

An optimal schedule for $P = 3$ is found by putting the first two tasks together in one stage, the third one by itself, and the last two ones in another stage — i.e., $(\mathbf{n}, \mathbf{p}) = ((2, 1, 2), (1, 1, 1))$. This solution is illustrated in Fig. 1b. In this case, $w(s_i, p_i) = 4,\ \forall i \in [1, 3]$, so $T^{-1} = 4$. Any other partition (or using fewer resources) would lead to a higher reciprocal throughput, so this solution is optimal.

An optimal schedule for $P = 6$ is illustrated in Fig. 1c. The optimal solution $(\mathbf{n}, \mathbf{p}) = ((1, 1, 2, 1), (1, 1, 2, 1))$ leads to a reciprocal throughput $T^{-1} = 3$. Notice that s_2 and s_3 have the same maximal weight in this solution. Using an additional resource in s_3 would reduce its weight. However, this would not improve the reciprocal throughput of the chain, so the optimal solution uses only $P' = 5$ resources. These optimal solutions can be found by OTAC, as shown in the next section.

IV. Optimal Scheduling using OTAC

We have discovered an optimal solution to the pipelined workflow scheduling problem described in Section III by extending a technique previously employed for the CCP problem [11]. As the CCP problem considers only stateful tasks, its solution cannot benefit from the replication of stateless tasks to improve throughput, whereas our new algorithm handles both kinds of tasks. We present our new algorithm in Section IV-A and its optimality proof in Section IV-B.

A. OTAC

OTAC (for *Optimal maximal-packing for TAsk Chains*) is based on two functions, `PROBE` and `SOLVE`. The latter finds a solution that minimizes the reciprocal throughput T^{-1} and the resources used P'. We explain them below.

`PROBE` is responsible for finding a solution with a reciprocal throughput equal or below T^{-1} for the linear chain of N tasks

Algorithm 1: PROBE(T^{-1})

Result: TRUE if there is a solution, FALSE otherwise
Output: T^{-1} updated if there is a solution, P' used resources, composition solution $(\mathbf{n}, \mathbf{p})$

1 $i \leftarrow 0$; */* current sub-sequence index */*
2 $b \leftarrow 0$, $e \leftarrow 1$; */* sub-sequence in $[b, e[$ */*
3 */* Loop to create packings */*
4 **while** $e \leq N$ **do**
5 $\quad i \leftarrow i + 1$, $b \leftarrow e$, $e \leftarrow b + 1$;
6 $\quad s_i \leftarrow \{\tau_b\}$, $n_i \leftarrow 1$;
7 $\quad$ */* Adds tasks to s_i while they fit */*
8 $\quad$ **while** $e \leq N$ *and* $w(s_i \cup \{\tau_e\}, 1) \leq T^{-1}$ **do**
9 $\quad\quad s_i \leftarrow s_i \cup \{\tau_e\}$, $n_i \leftarrow n_i + 1$, $e \leftarrow e + 1$;
10 $\quad$ **end**
11 $\quad$ */* Resources necessary for a packing */*
12 $\quad p_i \leftarrow \left\lceil \frac{w(s_i, 1)}{T^{-1}} \right\rceil$;
13 $\quad$ */* Has not fitted all tasks and s_i is stateless*/*
14 $\quad$ **if** $e \leq N$ *and* $s_i \cap \mathcal{T}_F = \emptyset$ **then**
15 $\quad\quad$ */* Searches for the first stateful tasks τ_f*
16 $\quad\quad$ *and adds other stateless tasks to s_i */*
17 $\quad\quad f \leftarrow e$;
18 $\quad\quad$ **while** $f \leq N$ *and* $\tau_f \in \mathcal{T}_L$ **do**
19 $\quad\quad\quad s_i \leftarrow s_i \cup \{\tau_f\}$, $n_i \leftarrow n_i + 1$, $f \leftarrow f + 1$;
20 $\quad\quad$ **end**
21 $\quad\quad$ */* Resources necessary for a packing */*
22 $\quad\quad p_i \leftarrow \left\lceil \frac{w(s_i, 1)}{T^{-1}} \right\rceil$;
23 $\quad\quad$ **if** $f > N$ **then** */* No more successors */*
24 $\quad\quad\quad e \leftarrow f$;
25 $\quad\quad$ **if** $e \neq f$ **then**
26 $\quad\quad\quad$ */* Successor sub-sequence */*
27 $\quad\quad\quad s_{\text{temp}} \leftarrow \tau_f$, $e \leftarrow f$;
28 $\quad\quad\quad$ */* Takes predecessor tasks */*
29 $\quad\quad\quad$ **while** $w(\{\tau_{e-1}\} \cup s_{temp}, 1) \leq T^{-1}$ **do**
30 $\quad\quad\quad\quad s_i \leftarrow s_i \setminus \{\tau_{e-1}\}$, $n_i \leftarrow n_i - 1$;
31 $\quad\quad\quad\quad s_{\text{temp}} \leftarrow \{\tau_{e-1}\} \cup s_{\text{temp}}$, $e \leftarrow e - 1$;
32 $\quad\quad\quad$ **end**
33 $\quad\quad\quad$ **if** $w(s_i, p_i - 1) \leq T^{-1}$ **then**
34 $\quad\quad\quad\quad p_i \leftarrow p_i - 1$;
35 $\quad\quad\quad\quad$ **while** $w(s_i \cup \{\tau_e\}, p_i) \leq T^{-1}$ **do**
36 $\quad\quad\quad\quad\quad s_i \leftarrow s_i \cup \{\tau_e\}$;
37 $\quad\quad\quad\quad\quad n_i \leftarrow n_i + 1$, $e \leftarrow e + 1$;
38 $\quad\quad\quad\quad$ **end**
39 $\quad\quad\quad$ **else** */* No benefit from taking tasks */*
40 $\quad\quad\quad\quad$ **while** $e \neq f$ **do**
41 $\quad\quad\quad\quad\quad s_i \leftarrow s_i \cup \{\tau_e\}$;
42 $\quad\quad\quad\quad\quad n_i \leftarrow n_i + 1$, $e \leftarrow e + 1$;
43 $\quad\quad\quad\quad$ **end**
44 **end**
45 $P' = \sum_{i=1,\ldots,k} p_i$; */* Number of used resources */*
46 **if** $P' \leq P$ **then** */* Checks if the solution is valid */*
47 $\quad T^{-1} \leftarrow \max_{i=1,\ldots,k} w(s_i, p_i)$;
48 $\quad$ **return** *(TRUE, T^{-1}, P', $(\boldsymbol{n}, \boldsymbol{p})$)*;
49 **else**
50 $\quad$ **return** *(FALSE, T^{-1}, P', $(\boldsymbol{n}, \boldsymbol{p})$)*;

using at most P resources. It achieves this solution by greedily packing the most tasks together in one stage while respecting T^{-1} and without wasting resources that could be better used in the next stage. Algorithm 1 shows how PROBE works in a loop (lines 4–44) creating stages.

While being computed, sub-sequence s_i starts at the task index b and ends at $e - 1$. The sub-sequence receives the most tasks while respecting T^{-1} (lines 5–10). It is assigned one resource, unless its first task is stateless and has a weight greater than T^{-1}, which would require more resources (line 12). We prevent any situations with a stateful task with a weight greater than T^{-1} in the SOLVE function, which will be explained later in Algorithm 2.

In the case that s_i is stateful, nothing more is to be done to it (line 14). For stateless sub-sequences, otherwise, our novel approach is employed (lines 17–44). Using the insight that the best solution for stateless tasks is to have them all in a single partition [12] using enough resources to respect T^{-1}, we try to pack the whole sequence of stateless tasks together (lines 17–22). As this new sub-sequence would require p_i resources (line 22), we check if it is possible to cover more tasks with them by limiting s_i to the stateless tasks that fit into $p_i - 1$ resources and leaving the remaining tasks for the next stateful sub-sequence s_{i+1}. This is done in three steps:

1) We try to construct s_{i+1} with the successor stateful task plus tasks from s_i, while respecting T^{-1} (lines 29–32).
2) If we are able to assign $p_i - 1$ resources to the new s_i and still respect T^{-1}, we add any additional consecutive tasks from s_{i+1} back in s_i that would still fit in $p_i - 1$ resources (lines 33–38).
3) Else, s_i remains with p_i resources and it receives back all stateless tasks that were taken by s_{i+1} in the previous steps (lines 39–43).

After creating all stages, we check if PROBE's solution requires at most P resources (lines 45–50). If the solution is valid, the value of $T(n, p)$ is returned in T^{-1} (line 48).

PROBE is employed by the SOLVE function in order to find an optimal solution. **SOLVE** (Algorithm 2) follows basically the same binary search procedure that would be applied for the CCP problem [11]. The single difference here comes in the form of its stop criterion (line 4). It has to take into account that the final reciprocal throughput may be fractional due to a division of the longest stage by up to P resources.

The binary search is done within the limits where an optimal solution can be found. The lower limit T^{-1}_{min} is based on a perfectly balanced pipeline (line 1), while the upper limit T^{-1}_{max} is affected by the task with the greatest weight (line 2). At each iteration (lines 3–11), PROBE checks if a solution exists for the middle value between these two limits (lines 4–5). If so, then all possible solutions above this value can be discarded from the search. Otherwise, all possible solutions below it can be discarded. The search continues until the stop criterion is met and the minimal reciprocal throughput is returned (line 12). Using this value, we can use the PROBE function to compute the composition of (N, P) with maximal throughput and minimal number of resources.

B. Proof of Optimality

Our optimality proof is built upon the proofs of previous solutions for the CCP problem [10], [11], [19], [20]. They have shown that our kind of solve function based on binary search always finds the minimal reciprocal throughput. Pinar and Aykanat [11] have shown that the optimal solution lies between the values of T^{-1}_{min} and T^{-1}_{max} that we employ. Meanwhile, Iqbal [19] has proven that our kind of greedy probe function always finds a solution for a given target reciprocal throughput, if one exists. The proof is based on the idea that, by creating stateful maximal packings (a concept explained in Section IV-B1), the least amount of work possible is left for the next stages. We extend this idea to also work with stateless maximal packings in order to complete our optimality proof.

Our proof is organized as follows. We first introduce the concepts of tight, full, and maximal packings in Section IV-B1, and we prove that maximal packings cover the most tasks with the fewest resources possible. We show in Section IV-B2 that `PROBE` always finds maximal packings. We show that maximal packings are unique in Section IV-B3 and we prove that, if solutions exist, there exists a unique solution based solely on maximal packings. Thus, the compositions found by `PROBE` are optimal, so our solution is optimal.

1) Packing Concepts:

Definition 1 (Packing)**.** *A stage (s_i, p_i) is considered a packing for the reciprocal throughput T^{-1} if and only if its duration is smaller than or equal to T^{-1}, i.e., $w(s_i, p_i) \leq T^{-1}$.*

Definition 2 (Tight packing)**.** *A packing (s_i, p_i) is considered tight for the reciprocal throughput T^{-1} if and only if no resource can be removed, i.e., $w(s_i, p_i - 1) > T^{-1}$.*

For the next definition, we assume that the sub-sequence s_i includes tasks $\{\tau_b, \ldots, \tau_{e-1}\}$ $(b < e)$.

Definition 3 (Full packing)**.** *A packing (s_i, p_i) is considered full for the reciprocal throughput T^{-1} if and only if it can include no more tasks, i.e., given successor $\tau_e \in \mathcal{T}$, $w(s_i \cup \{\tau_e\}, p_i) > T^{-1}$.*

Algorithm 2: `SOLVE`(N, P)

Result: Minimal reciprocal throughput T^{-1}

1 $T^{-1}_{\min} \leftarrow \frac{1}{P} \sum_{\tau \in \mathcal{T}} w_\tau$;
2 $T^{-1}_{\max} \leftarrow T^{-1}_{\min} + \max_{\tau \in \mathcal{T}} w_\tau$;
3 **while** $T^{-1}_{\max} - T^{-1}_{\min} \geq \frac{1}{P}$ **do**
4 $\quad T^{-1}_{\text{mid}} \leftarrow \frac{T^{-1}_{\max} + T^{-1}_{\min}}{2}$;
5 $\quad done,\ T^{-1}_{\text{mid}},\ P,\ (\mathbf{n}, \mathbf{p}) \leftarrow$ `PROBE`(T^{-1}_{mid}); /* *TRUE if a valid solution is found* */
6 $\quad$ **if** *done* **then**
7 $\quad\quad T^{-1}_{\max} \leftarrow T^{-1}_{\text{mid}}$; /* T^{-1} *can only get smaller* */
8 $\quad$ **else**
9 $\quad\quad T^{-1}_{\min} \leftarrow T^{-1}_{\text{mid}}$; /* T^{-1} *can only get bigger* */
10 $\quad$ **end**
11 **end**
12 $T^{-1} \leftarrow T^{-1}_{\max}$;

Definition 4 (Maximal packing)**.** *A packing (s_i, p_i) is considered maximal for the reciprocal throughput T^{-1} if and only if it is full, tight, and it meets one of the following criteria:*

1) *s_i is stateful;*
2) *s_i is stateless and has no successor; or*
3) *s_i is stateless and there exists a successor stateful subsequence s_{i+1} such that $(s_{i+1}, 1)$ is a maximal packing.*

Theorem 1.1. *A maximal packing (s_i, p_i) for reciprocal throughput T^{-1} always includes the largest number of tasks possible using the smallest number of resources possible.*

Proof. We split the proof for stateful and stateless subsequences. For stateful sub-sequences, Definition 3 means that the sub-sequence includes the largest number of tasks possible. Meanwhile, Definition 2 and Eq. (1) make it that the stage needs only one resource, the smallest number possible.

For stateless sub-sequences, Definitions 3 and 2 specify that no more tasks can be included nor resources removed. What remains to be proven is that including more tasks while also including additional resources does not improve the solution. This can be proven by considering three different scenarios for the successor sub-sequence:

1) There is no successor sub-sequence, which is trivial.
2) There is a stateless sub-sequence, which is impossible. By Definition 4, a stateless maximal packing only exists if its successor is stateful or does not exist. This happens because the optimal solution for a sequence of stateless tasks is to replicate them using the available resources, not to split them into multiple sub-sequences [12].
3) There is a stateful sub-sequence s_{i+1} such that $(s_{i+1}, 1)$ is a maximal packing. Together, they include $n_i + n_{i+1}$ tasks using $p_i + 1$ resources. If $m < n_{i+1}$ stateless tasks were to be moved from s_{i+1} to s_i, then at least $p_i + 1$ resources would be necessary to pack $n_i + m$ tasks (Definition 3). This is less tasks than before $(m < n_{i+1})$. Furthermore, if both sub-sequences were to be merged into one, then the merged sub-sequence would be stateful and it would not be a packing anymore.

Therefore, stateless maximal packings always include the largest number of tasks possible using the smallest number of resources possible. □

2) Maximal Packings in `PROBE`:

Theorem 1.2. *`PROBE` always finds maximal packings.*

Proof. By Definition 4, maximal packings have to be tight, full, and respect one of three other conditions.

Any stage (s_i, p_i) computed by `PROBE` has a duration $w(s_i, p_i) \leq T^{-1}$. This is ensured by the tests in lines 8, 33, and 35, and by the values of p_i computed in lines 12, 22, and 34, so the stage is a packing. Additionally, the value of p_i cannot be reduced, so the packing is tight (Definition 2).

The tests in lines 8, 18, 35, and 40 ensure that no more tasks can be added to the packing, so it is full (Definition 3).

Finally, the three final conditions for a maximal packing are covered as follows:

1) We verify if s_i is stateful in line 14.
2) We see if s_i is stateless and has no successor in line 23.
3) We check if s_i is stateless and there exists a successor maximal packing $(s_{i+1}, 1)$ in lines 25–43. s_{i+1} is ensured to be stateful by the tests in lines 18 and 25 when computing s_i. It is ensured to be maximal in the next iteration of the outer loop (lines 4–44) (s_{i+1} meets condition (1) to be maximal).

Consequently, all packings computed by PROBE are maximal. □

3) Maximal Packing Solution Uniqueness: Given a starting task τ_b and a reciprocal throughput T^{-1}, there exists a single maximal packing (s_i, p_i) with $s_i = \{\tau_b, \ldots, \tau_{e-1}\}$. This comes naturally from Theorem 1.1. It proves that a maximal packing always includes the largest number of tasks possible using the smallest number of resources possible. For instance, it would be absurd to have two different solutions with these properties. This idea is employed in Theorem 1.3.

Theorem 1.3. *If there are valid solutions for (N, P) and a target T^{-1}, then there exists a unique solution (n^*, p^*) with $T(n^*, p^*) \leq T^{-1}$ composed only of maximal packings and that uses a minimal number of resources.*

Proof. First, if all stages are maximal packings, then, by definition, all stages have $w(s_i, p_i) \leq T^{-1}$, so $T(n^*, p^*) \leq T^{-1}$.

Second, for each task τ_b starting a stage, there is only a single maximal packing. So the solution is unique.

Third and final, assume there is another solution (n', p') composed only of maximal packings that requires fewer resources than (n^*, p^*). They must differ in one of two ways:

- (n', p') has a stateless stage (s_i, p'_i) with fewer resources than the equivalent (s_i, p_i) in (n^*, p^*); or
- They have stages starting at different tasks.

The first scenario is a contradiction because (s_i, p_i) must be tight to be maximal. So there cannot be a solution that takes less than p_i resources for stage s_i. The second scenario is also a contradiction. Indeed, for a stage to start at a different task in both solutions requires a previous stage to have finished at a different task. Moreover, there is only a single maximal packing starting at a given task. So one of the solutions is not composed only of maximal packings. Consequently, the solution is unique and uses a minimal number of resources. □

By combining Theorems 1.2 and 1.3, we can conclude that our PROBE function always finds optimal compositions. Again, as our kind of solve function based on binary search always finds the minimal (i.e., optimal) reciprocal throughput, we can conclude that the solution found by OTAC to this pipelined workflow scheduling problem is optimal.

4) Complexity: In the PROBE function, we can notice that no task will be considered more than once at the initial part of the main loop (lines 4–12 in Algorithm 1). When handling a stateless sub-sequence, the additional tasks will only be added or removed from the sub-sequence a constant number of times. They will not be considered again later. Thus, the main loop (lines 4–44) computes $O(N)$ operations. As $k \leq N$, the loop on line 45 also computes $O(N)$ operations, so the final time complexity of PROBE is in $O(N)$.

The SOLVE function calls PROBE until its stop criterion is met. Given that the range of values considered is equal to $w_{\max} = \max_{\tau \in \mathcal{T}} w_\tau$, and that the loop stops when their difference is smaller than $\frac{1}{P}$, this loop is executed $O(\log w_{\max} + \log P)$ times. Combined with the complexity of PROBE, the time complexity of SOLVE is in $O(\log(w_{\max} P)N)$.

5) Extension for Weight Variations: Due to its low complexity, it is possible to adapt OTAC in cases where task weights change in time, for instance when the signal rate ratio changes. Starting with an optimal configuration computed by OTAC, the approach simply consists in detecting that the solution is no longer optimal and recomputing an optimal one.

To detect that a schedule is no longer optimal, it is sufficient to test if the current stages are still tight and full packings (Definitions 2 and 3). Indeed, Theorem 1.3 shows that the optimal solution can only be defined by maximal packings, using a minimal number of resources. When the weight of a task changes in time, the initial maximal packing may no longer be a packing, tight, or full. Checking these properties can be done in $O(N)$ during execution. By checking the current solution repeatedly, we may trigger OTAC whenever we need a new solution. The practical evaluation of this approach is left for future work.

V. Experimental Evaluation

With the interest of fully exploring OTAC's capacity to optimize the throughput of communication chains, while minimizing resource usage, we have organized an experimental evaluation in two parts. In the first part, we explore the effects of scheduling a thousand different synthetic task chains using simulation. In the second part, we present a concrete use case using a Digital Video Broadcasting – Second Generation (DVB-S2) implementation [17] on AFF3CT [7]. We detail our workloads, tested scheduling algorithms, and experimental environment in Sections V-A, V-B, and V-C, respectively. Section V-D covers our simulation results, while Section V-E presents the results for the DVB-S2 communication chain.

A. Workloads

For the **simulation experiments**, we have generated a thousand synthetic task chains of 20 tasks each. Each task has a weight in μs taken randomly from a uniform distribution in the interval $[100, 35000]$. This interval was chosen based on performance measurements over the original DVB-S2 chain.

For each chain, we also create variations by changing their **stateless ratio (SR)** between zero and one, where zero (resp., one) means all tasks are stateful (resp., stateless). An increase in stateless ratio of 0.1 means that two tasks are changed from stateful to stateless. We explore the schedule of all these task chains over $P = \{5, 10, 15, 20, 25\}$ resources in Section V-D.

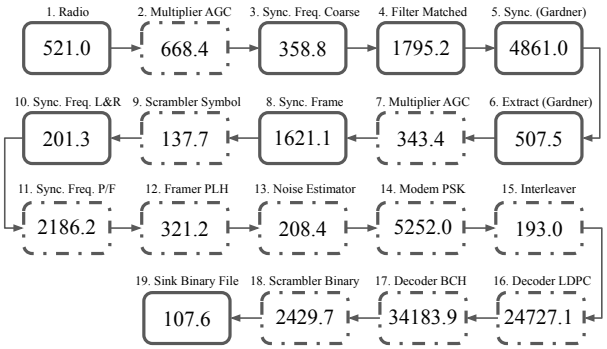


Fig. 2: Illustration of the DVB-S2 reception chain. Each box represents a task. Task names (adapted from [30]) and task weights (rounded values in μs) are given with each task. Solid (resp., dashed) lines represent stateful (resp. stateless) tasks.

Our **concrete use case** is based on a preexisting open-source implementation of the **DVB-S2** communication chain [31]. DVB-S2 is a standard used for satellite transmission of video contents [17]. Thanks to its wide range of channel coding and modulation options, the DVB-S2 standard can be used for broadcasting services and television transmission.

Our evaluation focuses on DVB-S2's reception chain, as forward error code decoding is the most compute-intensive part of its process. We illustrate this chain in Fig. 2. The task weights were measured in our experimental platform for a frame interleaving $I = 16$ (16 frames are processed by each task during each step) and $K = 14232$ information bits transmitted per frame (MODCOD 2) [30]. The signal ratio is fixed to 4dB. Each execution decodes 1.28 GB of data. We also consider a pipelined and replicated schedule of the reception chain as a *baseline* for comparison. It was manually optimized for a different multicore machine [30].

B. Scheduling Algorithms

We have chosen to employ algorithms from the state of the art (Nicol) or to adapt greedy approaches seen in practical contexts (see Section II-B) in order to evaluate the benefits of OTAC. We have avoided the ILP approaches proposed in the context of StreamIt due to their complexity and their difference in constraints (DAGs instead of chains, grouping together non-contiguous tasks). The chosen schedulers are the following:

- **Nicol** [11], [20]: optimal algorithm when only pipelining is possible (does not consider replication).
- **RPT**: scheduler that tries to set one resource per task. If $P < N$, it greedily fuses together the neighboring stages that have the smallest combined weight.
- **RB**: recursive-bipartitioning-based scheduler. It partitions the chain into two parts with similar weight, and provides a proportional part of the resources to each partition ($\lfloor \frac{P}{2} \rfloor$ and $\lceil \frac{P}{2} \rceil$). It stops when a partition contains a single task or resource. A stage with a single stateless task can benefit from multiple resources for replication.
- **RPTm**: as RPT, but neighboring stateless stages are merged together at the end of the algorithm to share the same pool of resources during replication.
- **RBm**: as RB with the merging optimization of RPTm.

As our results have shown us that the schedulers with the merging optimizations (RPTm and RBm) always outperform their standard counterparts, we fill focuses our analysis on OTAC, Nicol, RPTm, and RBm only.

C. Hardware and Software Environments

Experiments were executed on a single server with two 18-core Intel Xeon Skylake Gold 6240 processors at 2.6 GHz (36 cores in total), 192 GB of DDR4 DRAM at 2933 MT/s, and 1 TB of local storage with a SATA Seagate ST1000NX0443 at 7.2krpm. The server runs CentOS release 7.6.1810 with kernel 3.10.0-957.el7.x86_64. We used AFF3CT v3.0.2, GCC v12.2.0, and Python v3.8.0. For the DVB-S2 experiments, threads were pinned using hwloc v2.7.0. No other users or applications were using the server during the experiments. Data was kept local to the server to avoid network interference.

D. Simulation Results

1) Optimal Solutions: Our simulation experiments aim at comparing schedulers in terms of throughput and resource utilization. Throughout these simulation with 1000 task chains, 11 stateless ratio values, and 5 different numbers of resources, **OTAC always obtained the maximum throughput with minimal resource utilization**. We summarize the 55000 schedules per algorithm in Fig. 3 in comparison to the optimal schedules of OTAC. Each row holds the results for a given scheduler. Each figure presents the results for a given P. Each column in a figure counts the number of **optimal** schedules, schedules with maximal throughput that use resources in **excess**, and **suboptimal** schedules, adding up to 1000 schedules (one per chain).

Fig. 3 provides several insights on the schedulers' behavior. RPTm finds more solutions with **maximal throughput** (*optimal* + *excess*) as the number of available resources increases. In general, it is easier to find the best solutions when fewer stages have to be merged due to a lack of resources. RPTm tends to find fewer maximal throughput solutions when SR increases. RPTm bases the number of replicas of a pipeline stage on the number of stateless tasks in it (instead of the tasks' weights), leading to suboptimal schedules.

RPTm found excess solutions for all task chains with SR $= 0$ and $P = 20$ or 25. Putting each of the 20 tasks on their own stage ensures that the highest stage weight is minimal in these cases. Nonetheless, these solutions were always using resources in excess because there were always neighboring stages of lower weight that could be fused together, freeing resources without affecting throughput. We can also see that RPTm always finds optimal solutions when all tasks are stateless (except when $P = 25$). This happens because it merges all stages into a single one replicated P times, which is known to be optimal [12]. This is not the case for $P = 25$ because RPTm uses only up to $P' = N = 20$ resources.

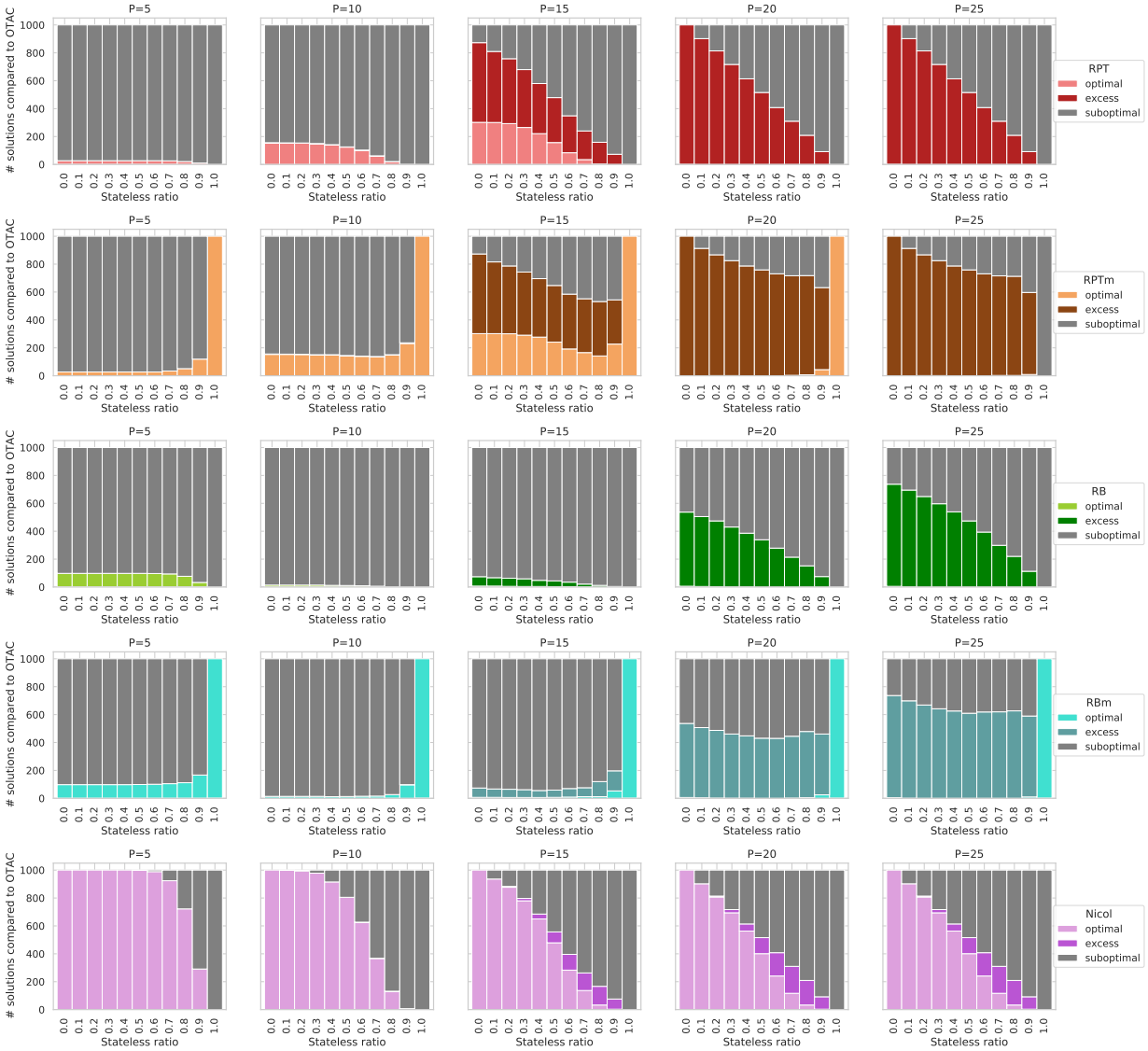


Fig. 3: Simulation: number of solutions of different qualities for different schedulers, numbers of processors, and stateless ratios compared to OTAC. *Optimal* means maximum throughput with minimum number of resources. *Excess* means maximum throughput with extra resources. *Suboptimal* means suboptimal throughput.

RBm behaves similarly to RPTm in many ways: merging improves performance, fewer maximal throughput solutions are found when SR increases, optimal solutions are found when all tasks are stateless. In the tested scenarios, RBm is less effective than RPTm in general, with the exception of the cases with very few resources ($P = 5$). Here, RBm is able to find 2074 optimal solutions (18.9% of the cases) whereas RPTm only finds 1393 (12.7%). These results are biased by the 1000 optimal solutions found when all tasks are stateless. Meanwhile, OTAC always finds optimal solutions.

Nicol shows a very different behavior. Firstly, Nicol tends to find optimal solutions less often when P increases. With more resources, the potential for performance gains through replication is higher, leading to more suboptimal solutions by Nicol. This also explains the decrease in optimal solutions found with the increase in SR (with zero optimal solutions for SR = 1.0). Still, in situations where pipelining is more important than replication (few resources or low SR), Nicol finds optimal solutions. For instance, for $P = 5$, Nicol finds all optimal solutions for SR < 0.5, and about 97.7% of them for scenarios in this SR range and $P = 10$.

Overall, these results quantify how difficult it can be to find an optimal schedule when pipelining and replication are involved, but they do not qualify how good or bad the other schedules are. We explore their differences in terms of resources used and throughput in the next sections.

2) Resources Used: We study the different numbers of resources used by OTAC, RBm, and Nicol for $P = 20$ and SR $= 0.5$ and 0.9 in Fig. 4. In these histograms, each bar counts the number of optimal, excess, and suboptimal solutions found

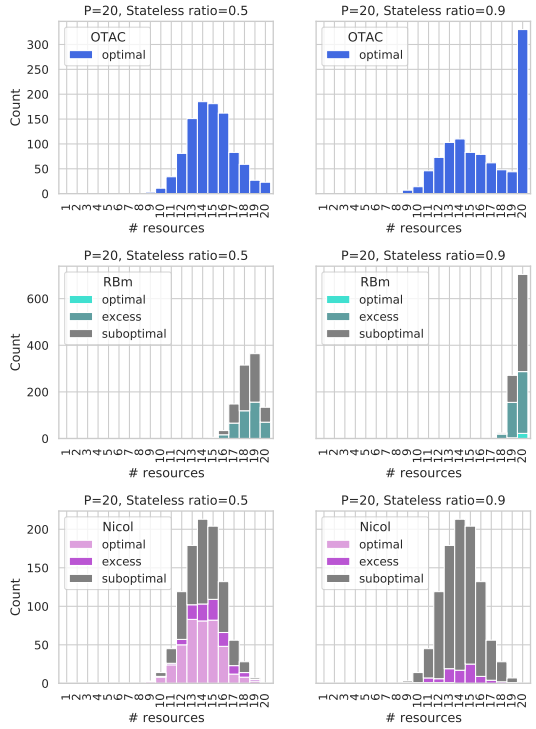


Fig. 4: Simulation: histograms of number of resources used (P') for the schedules computed by OTAC, RB, RBm, and Nicol for $P = 20$ and SR $= 0.5$ (left) and 0.9 (right). Vertical axes at different scales for different schedulers.

with a given number of resources. We chose two different stateless ratios to illustrate how they affect solutions (i.e., the more stateless tasks, the more resources the task chain can benefit from). RPTm solutions are not illustrated. In these circumstances, they consistently use 20 resources.

When considering the solutions obtained with OTAC, we can notice that the majority of solutions require fewer than 20 resources. For SR $= 0.5$ (resp., 0.9), OTAC uses an average of 14.8 (resp., 16.3) resources, with only 23 (resp., 330) schedules using all 20 resources available. When the SR is higher, OTAC profits from the higher replication potential and ends up using more resources for a higher throughput.

RBm, meanwhile, tends to use many more resources than necessary in its schedules. For SR $= 0.5$, it uses an average of 18.4 resources, while only achieving 431 maximal throughput solutions. This represents about 3.6 (24.3%) more resources than OTAC. For SR $= 0.9$, these values increase to an average of 19.7 resources for 461 maximal throughput solutions.

Nicol, for its part, tends to underutilize resources. As shown in Fig. 4, it computes the same schedule independently from the task chains' SR. As SR increases, more of these schedules become suboptimal. On average, Nicol uses 14.2 resources here. This leads to 516 maximal throughput solutions for SR $= 0.5$, but only 92 such solutions for SR $= 0.9$.

These results reinforce the importance of scheduling with the right number of resources: too many resources lead to wastage, while too few lead to suboptimal throughput.

3) Comparing Throughput: Considering that each task processes only one frame at a time (differently from DVB-S2), we explore the throughput achieved for the task chains with SR $= 0.5$ in two scenarios: with $P = 10$ and $P = 20$. We illustrate these results in Fig. 5, where the figures on the left represent the throughput of OTAC in frames per second (FPS), while the other figures represent the throughput difference between OTAC and other schedulers when considering suboptimal schedules only[2]. We can see that the average throughput obtained by OTAC increases when more resources are available (from 223 to 315 FPS — a factor of 1.41). While it uses almost all resources for scheduling tasks when $P = 10$, the situation changes when $P = 20$ (see Fig. 4). As schedules become more limited by the stateful tasks' weights, the results become more concentrated to the left of the histogram.

RBm finds the fewest maximal throughput solutions in both scenarios, while showing a broad distribution of performance losses, with an average of 57.17 and 62.38 fewer FPS than OTAC for $P = 10$ and 20, respectively, when only considering suboptimal solutions. These differences improve to 56.42 and 35.49 fewer FPS when all solutions are considered, which is still 11% fewer FPS in the best of cases.

Nicol found 804 optimal solutions when $P = 10$, leading to a low count of suboptimal solutions. It achieved an average of 14.36 fewer FPS than OTAC on its suboptimal solutions, and an average of 2.80 fewer FPS over all schedules. Yet, these differences increased to 31.41 and 15.20 when $P = 20$ (about 5% fewer than OTAC), showing the importance of replication.

Finally, RPTm shows an improving performance with the increase in the number of resources. Its solutions showed an average of 21.52 fewer FPS than OTAC for suboptimal solutions and of 18.40 fewer FPS over all solutions for $P = 10$. These differences change to 31.50 and 7.59 fewer FPS over the respective solutions when $P = 20$. Although the range and magnitude of the performance differences increase when more resources are available, more solutions achieve maximal throughputs (from 145 to 759 in these cases), leading to a smaller average difference. Still, in the case of RPTm, these improvements come at the cost of resource wastage.

Overall, **our simulation results have highlighted how arbitrarily easy or difficult it can be for other schedulers to find optimal solutions**. The quality of their solutions strongly depends on the characteristics of the task chain and the number of resources available, while OTAC is able to handle all scenarios optimally. We explore how these points translate to a real communication chain in the next section.

E. DVB-S2 Standard Results

Fig. 6 shows the results obtained for DVB-S2's reception chain. Full lines represent the simulated maximum throughput each scheduler could achieve based without overhead. Lines with dots represent the average throughput obtained with different numbers of resources over ten executions. A few

[2]Maximal throughput schedules would only increase the bar at zero.

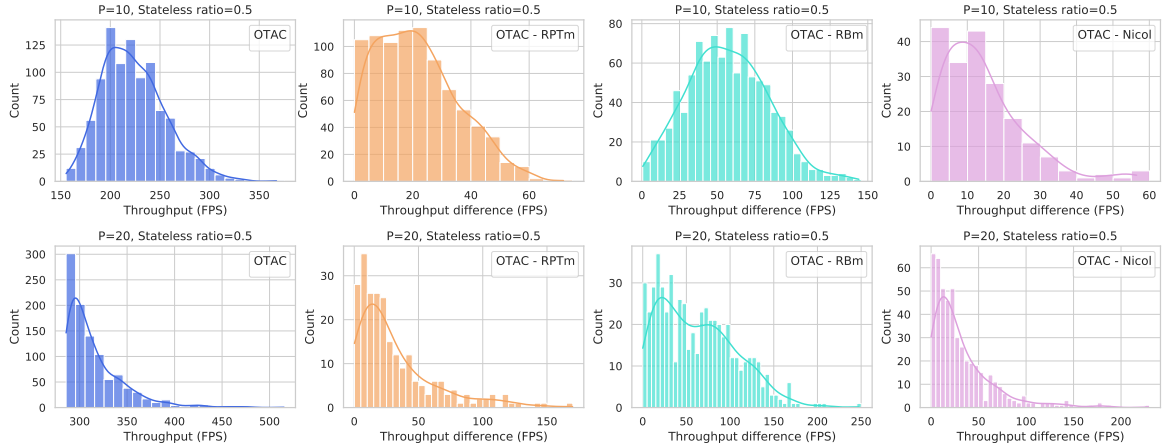


Fig. 5: Simulation: histograms of throughput in terms of frames per second achieved by OTAC (left) for $P = 10$ and $P = 20$ and SR $= 0.5$; throughput difference between OTAC and RPTm, RBm, and Nicol for the instances where they find suboptimal schedules. Axes at different scales depending on the histogram.

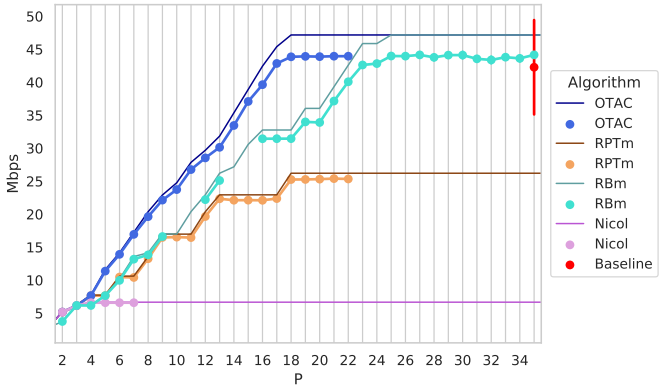


Fig. 6: DVB-S2: throughput in Mbps as a function of the number of available resources for different schedulers.

(**n**, **p**)	Sim. T	T (avg)	T (med)	P / P'
OTAC				
((4,1,5,7,2), (1,1,1,14,1))	47.17	43.86	43.95	18 / 18
RPTm				
((1,1,1,1,1,1,1,1,2,8,1), (1,1,1,1,1,1,1,1,1,8,1))	26.21	25.26	25.23	18 / 18
RBm				
((1,2,1,1,1,1,1,2,8,1), (1,1,1,1,1,1,1,1,15,1))	47.17	43.98	44.01	25 / 24
Nicol				
((15,1,1,2), (1,1,1,1))	6.66	6.58	6.58	4 / 4
Baseline				
((1,3,1,3,3,2,5,1), (1,1,1,1,1,1,28,1))	47.17	42.3	44.58	35 / 35

TABLE I: DVB-S2: Best schedules, throughput (Mbps), and number of resources for the different algorithms.

points are missing for RBm ($P = \{10, 11, 14, 15, 16\}$) because its schedules were not achievable. Some successive tasks in DVB-S2 must remain in the same stage of the pipeline. But, as this constraint is not visible at the scheduler level, RBm is unable to enforce this rule. Additionally, as the maximum throughput achievable by all schedulers plateaus after a certain number of resources (e.g., at $P = 18$ for OTAC), we limited our experiments to fewer than 36 resources in most cases to avoid occupying the server unnecessarily.

Fig. 6 shows that OTAC and RBm achieve the highest throughput, while RPTm and Nicol find suboptimal schedules. The hand-tuned baseline achieves a performance similar to OTAC but requires almost twice more resources. We can also notice a 7% gap between the best expected throughput (47.17 Mbps) and the one achieved by OTAC with $P = 18$ (43.86 Mbps, on average). Among the sources of this gap is the synchronization overhead of managing a pipelined and replicated task chain, not explicitly considered in our model.

Table I provides more details about the best results achieved by the different algorithms (highest throughput with minimal number of resources used). We include here their simulated throughput, and also their average and median values. Nicol achieves the lowest throughput by not replicating the task with the highest weight (34183.9 μs in Fig. 2). Meanwhile, OTAC, RBm and the baseline are limited by the highest weight among *stateful* tasks (4861 μs), leading to a 7× higher throughput.

When comparing the best solutions found by the algorithms using the Mann-Whitney U test with 5% confidence, we cannot say that OTAC and RBm perform differently (p-value $= 0.73$), in spite of RBm using 6 more resources (33% in excess). OTAC and the baseline show statistically different results (p-value $= 0.009$), with the baseline achieving a median throughput 0.6 Mbps above OTAC. As OTAC only replicates the fourth pipeline stage enough so that its weight (4790.8 μs) is below the one of its second stage, we believe its solution to be more sensitive to fluctuations in task latency and replication overhead. However, this difference may not justify the 17

additional resources used by the baseline.

VI. Concluding Remarks

In this article, we considered the problem of scheduling pipelined and replicated task chains on homogeneous computing resources for SDR applications. Based on the characteristics of these tasks (stateless or stateful), we have proposed OTAC, a scheduler that maximizes throughput while minimizing resource usage. We have shown that OTAC finds solutions in $O(\log(w_{\max}P)N)$, and we have proved its optimality.

Through an extensive experimental evaluation using both simulation and the DVB-S2 communication standard, we have shown how OTAC outperforms other strategies. Our simulation results have highlighted how arbitrarily easy or difficult it can be for other schedulers to find optimal solutions, while OTAC always does so. The experiments with DVB-S2 have shown that OTAC achieves almost 44 Mbps over 18 cores, which is 7% from the limit without overhead, while it uses down to 51% of the resources of a hand-tuned solution.

For SDR systems, OTAC focuses on pipelined workflow scheduling problems for linear task chains, a limited aspect of these systems. It assumes that resources are identical, so it cannot benefit from more heterogeneous platforms. In addition, OTAC tries to avoid communication overhead (by keeping neighboring tasks together), but it does not consider them explicitly. Finally, OTAC has been evaluated as an offline method, but we have shown that a more dynamic approach for dynamic weight remains possible, combining a detection of non-optimality with a restart of OTAC.

We thus plan to investigate how to adapt OTAC (or propose new solutions) to different shapes of task graphs and for more heterogeneous platforms (e.g., uniform resources with different clock frequencies or platforms with accelerators). We aim to integrate communication weights in our model based on communication overhead profiled for different levels of task replication, and to see how to use them within OTAC. Finally, we will study the problem of scheduling multiple task chains over shared resources with different optimization objectives (e.g., maximum sum of throughput, max-min fairness).

References

[1] M. Palkovic, J. Declerck, P. Raghavan, A. Dejonghe, and L. Van der Perre, "Dart - a high level software-defined radio platform model for developing the run-time controller," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2011.

[2] ETSI 3GPP - TS 38.212, "Multiplexing and Channel Coding," 2018.

[3] P. Giard, G. Sarkis, C. Leroux, C. Thibeault, and W. J. Gross, "Low-Latency Software Polar Decoders," *J. Signal Process. Syst.*, vol. 90, no. 5, may 2018.

[4] B. Le Gal and C. Jego, "High-Throughput Multi-Core LDPC Decoders Based on x86 Processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, 2016.

[5] M. R. Maheshwarappa, M. Bowyer, and C. P. Bridges, "Software Defined Radio (SDR) architecture to support multi-satellite communications," in *2015 IEEE Aerospace Conference*, 2015, pp. 1–10.

[6] A. Karlsson, J. Sohl, J. Wang, and D. Liu, "ePUMA: A unique memory access based parallel DSP processor for SDR and CR," in *2013 IEEE Global Conference on Signal and Information Processing*, 2013.

[7] A. Cassagne, R. Tajan, O. Aumage, C. Leroux, D. Barthou, and C. Jégo, "A DSEL for high throughput and low latency software-defined radio on multicore CPUs," *Concurrency and Computation: Practice and Experience*.

[8] M. Léonardon, A. Cassagne, C. Leroux, C. Jégo, L.-P. Hamelin, and Y. Savaria, "Fast and Flexible Software Polar List Decoders," *J. Signal Process. Syst.*, vol. 91, no. 8, p. 937–952, aug 2019. [Online]. Available: https://doi.org/10.1007/s11265-018-1430-3

[9] M. Kudlur and S. Mahlke, "Orchestrating the Execution of Stream Programs on Multicore Platforms," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: Association for Computing Machinery, 2008.

[10] S. Bokhari, "Partitioning problems in parallel, pipeline, and distributed computing," *IEEE Transactions on Computers*, vol. 37, no. 1, 1988.

[11] A. Pınar and C. Aykanat, "Fast optimal load balancing algorithms for 1D partitioning," *Journal of Parallel and Distributed Computing*, vol. 64, no. 8, 2004.

[12] A. Benoit and Y. Robert, "Complexity results for throughput and latency optimization of replicated and data-parallel workflows," *Algorithmica*, vol. 57, no. 4, 2010.

[13] Y. Yi, W. Han, X. Zhao, A. T. Erdogan, and T. Arslan, "An ILP formulation for task mapping and scheduling on multi-core architectures," in *Design, Automation & Test in Europe Conference & Exhibition*, 2009.

[14] W. Che, A. Panda, and K. S. Chatha, "Compilation of stream programs for multicore processors that incorporate scratchpad memories," in *Design, Automation & Test in Europe Conference & Exhibition*, 2010.

[15] N. Gayen, J. Ax, M. Flasskamp, C. Klarhorst, T. Jungeblut, M. Tang, and W. Kelly, "Scalable Mapping of Streaming Applications onto MPSoCs Using Optimistic Mixed Integer Linear Programming," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2018.

[16] B. Bloessl, M. Müller, and M. Hollick, "Benchmarking and Profiling the GNU Radio Scheduler," in *9th GNU Radio Conference (GRCon 2019)*. Huntsville, AL: GNU Radio Foundation, September 2019.

[17] ETSI EN 302 307 V1.2.1, "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)," 2009.

[18] A. Benoit, U. V. Çatalyürek, Y. Robert, and E. Saule, "A Survey of Pipelined Workflow Scheduling: Models and Algorithms," *ACM Comput. Surv.*, vol. 45, no. 4, aug 2013.

[19] M. A. Iqbal, "Approximate algorithms for partitioning and assignment problems," Tech. Rep., 1986.

[20] D. Nicol, "Rectilinear Partitioning of Irregular Data Parallel Computations," *Journal of Parallel and Distributed Computing*, vol. 23, no. 2, 1994.

[21] Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, p. 406–471, dec 1999.

[22] N. Vydyanathan, U. V. Catalyurek, T. M. Kurc, P. Sadayappan, and J. H. Saltz, "Toward Optimizing Latency Under Throughput Constraints for Application Workflows on Clusters," in *Euro-Par 2007 Parallel Processing*, A.-M. Kermarrec, L. Bougé, and T. Priol, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.

[23] G. Radio. (2023) GNU Radio – the Free and Open Software Radio Ecosystem. [Online]. Available: https://github.com/gnuradio/gnuradio

[24] J. Mack, S. Gener, A. Akoglu, J. Holtom, A. Chiriyath, C. Chakrabarti, D. Bliss, A. Krishnakumar, A. Goksoy, and U. Ogras, "GNU Radio and CEDR: Runtime Scheduling to Heterogeneous Accelerators," in *Proceedings of the GNU Radio Conference*, vol. 7, no. 1, 2022.

[25] J. Morman, M. Lichtman, and M. Müller, "The Future of GNU Radio: Heterogeneous Computing, Distributed Processing, and Scheduler-as-a-Plugin," in *IEEE Military Communications Conference*, 2022.

[26] S. A. William Thies, Michal Karczmarek, "StreamIt: A Language for Streaming Applications," *Lecture Notes in Computer Science*, 2002.

[27] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, "A Stream Compiler for Communication-Exposed Architectures," *SIGPLAN Not.*, vol. 37, no. 10, oct 2002.

[28] C. Glitia, P. Dumont, and P. Boulet, "Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing," *Multidim. Systems and Signal Processing*, vol. 21, 2010.

[29] A. Cassagne. (2023) AFF3CT A Fast Forward Error Correction Toolbox. [Online]. Available: https://github.com/aff3ct/aff3ct
[30] A. Cassagne, M. Léonardon, R. Tajan, C. Leroux, C. Jégo, O. Aumage, and D. Barthou, "A flexible and portable real-time DVB-S2 transceiver using multicore and SIMD CPUs," in *2021 11th International Symposium on Topics in Coding (ISTC)*, 2021.
[31] A. Cassagne, M. Léonardon, R. Tajan, and C. Leroux. (2023) DVB-S2 SDR transceiver. [Online]. Available: https://github.com/aff3ct/dvbs2