Multi-resource scheduling for FPGA systems

Renaud Pacalet, Matteo Bertolino, Ludovic Apvrille, Andrea Enrici

Multi-Resource Scheduling for FPGA Systems

Matteo Bertolino, Renaud Pacalet, Ludovic Apvrille

LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Andrea Enrici

Nokia Bell Labs, Centre de Villarceaux, Nozay, France

Abstract

In modern cloud data centers, reconfigurable devices (FPGAs) are used as an alternative to Graphics Processing Units to accelerate data-intensive computations (e.g., machine learning, image and signal processing). Currently, FPGAs are configured to execute fixed workloads, repeatedly over long periods of time. This conflicts with the needs, proper to cloud computing, to flexibly allocate different workloads and to offer the use of physical devices to multiple users. This raises the need for novel, efficient FPGA scheduling algorithms that can decide execution orders close to the optimum in a short time. In this context, we propose a novel scheduling heuristic where groups of tasks that execute together are interposed by hardware reconfigurations. Our contribution is based on gathering tasks around a high-latency task that hides the latency of tasks, within the same group, that run in parallel and have shorter latencies. We evaluated our solution on a benchmark of 37500 random workloads, synthesized from realistic designs (i.e., topology, resource occupancy). For this testbench, on average, our heuristic produces optimum makespan solutions in 47.4% of the cases. It produces acceptable solutions for moderately constrained systems (i.e., the deadline falls within 10% of the optimum makespan) in 90.1% of the cases.

Keywords: Resource Constrained Scheduling, FPGA, Reconfigurable Hardware

1. Introduction

In the last decade, the use of Field Programmable Gate Arrays (FPGAs) has steadily increased to accelerate the computation of data-intensive workloads (e.g., machine learning, financial applications, image and signal processing). Specifically in the domain of cloud computing, the use of FPGAs has received much attention. As defined by the U.S. National Institute of Standards and Technology, cloud computing "[...] is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [1]. The economy of cloud computing is essentially based on the idea that "using 1000 servers for one hour costs no more than using one server for 1000 hours" [2].

In cloud infrastructures, FPGAs are an interesting alternative to Graphics Processing Units (GPUs): FPGAs can offer significant speed-up in the execution of many different types of workloads and are also power-efficient [3]. In existing cloud infrastructures (e.g., Microsoft Azure, Amazon Web Services - AWS), FPGAs are used as fixed accelerators, where to statically deploy workloads that are designed once and executed repeatedly over time. This conflicts with the need of cloud computing to deploy different workloads at run-time and to offer to multiple users the shared use of physical devices. This requirement raises the need for efficient scheduling algorithms that can rapidly take decisions while accounting for the shared use of FPGA resources (e.g., logic elements, memory blocks) in both space and time. In this paper, we present a novel scheduling heuristic called *Slot* that aims to reduce the an application's makespan (source-to-sink execution latency). The rationale of Slot is to iteratively transform an input dependency graph of tasks (i.e., units of work), which allows for multiple execution orders, into a totally ordered graph of groups of tasks. At each iteration, we create a group of tasks, called a slot, by contracting task nodes in the current graph. Tasks are grouped, by considering tasks' resource needs,

around a high-latency task that executes in parallel to lower-latency tasks, thus hiding their latency. Slots are sequentially executed, interposed by FPGA total reconfigurations; tasks in slot n start execution when all tasks in preceding slots have completed.

In the rest of this paper, Section 2 provides the reader with a summary of relevant knowledge from the context of our work: FPGA virtualization and multi-tenant FPGAs used in cloud architectures. This section also un-ambiguously states the basic definitions for our research problem. Section 3 discusses our research problem and positions our solution with respect to existing works related to FPGA scheduling. Section 4 presents our scheduling heuristic. Section 5 reports a detailed evaluation of the heuristic's run-time and quality of solutions. Additionally, we compare solutions from our heuristic to both optimal scheduling and to solutions produced by well-known heuristics from the literature. Section 6 concludes this paper and presents directions for future works. We include three appendixes that present relevant work for our evaluation in Section 5. In Appendix A we detail our random generator of FPGA scheduling problems. We present an exact formulation of our scheduling problem in Appendix B, that we used to produce optimal solutions for comparison. In Appendix C we correct the pseudo-code of a concurrent heuristic from the literature, HPF-NF in [4].

2. Context

To optimize FPGA utilization and return-on-investment in cloud computing, the fabric of a single FPGA device is shared among multiple users thanks to the FPGA's partial reconfiguration capacibilities [5]. Partial reconfiguration allows to reconfigure the functionality of a pre-defined region of the FPGA while network services (tasks) are deployed. These regions are typically called Partial Reconfigurable Regions (PRRs) and have led to the notion of multi-tenant FPGAs. FPGA sharing (also called multiplexing or partitioning) can occur either temporally or spatially. Temporal sharing refers to scenarios where the FPGA device fabric (or one of its PRRs) is allocated to different users or tenants at

different time slots. This is the more conventional scenario that is most widely used nowadays in the industry. Recently, researchers have started to explore the scenario of spatial sharing, where different tenants' tasks are simultaneously located on different isolated PRRs within the same FPGA device. While possible in principle, this scenario raises a security and privacy concerns [6]. As such it is currently under investigation and is not yet a reality.

Temporal sharing of FPGAs is possible thanks to partial reconfiguration. The latter enables dynamically reconfiguring a portion of the FPGA, while the rest of the device logic continues to operate [7]. An FPGA is partitioned into a static region and one or more PRRs. These PRRs can be each configured with its own bitstream without affecting other PRRs. In this configuration, PRRs communicate with the host CPU or other peripherals via pre-defined interfaces implemented in the static region.

In the context of multi-tenant FPGAs, scheduling of hardware tasks is intrinsically relevant to other aspects of FPGA design (e.g., security, serviceability, billing). We highlight to the reader that, in our work, we focus on scheduling in the context of an FPGA being temporally shared among tenants, regardless of the entity that demands to execute hardware tasks (e.g., system designer, network orchestrator). The reason for this choice is to enlarge the spectrum of possible users of our contribution that is not specific to cloud FPGAs but also applies to FPGAs in other domains (e.g., embedded systems).

To help the reader in understanding how our work relates to other research efforts in multi-tenant FPGAs, we position our work with respect to [8, 9]. In [8], the authors present a mapper that assigns hardware tasks called Partial Reconfiguration Modules (PRMs) onto PRRs. The goal of this PRR-to-PRM mapper is to map one PRM to as many PRRs as possible in order to maximize the chances that, when a given PRM is requested by a user, the hosting PRR is effectively available. This minimizes the users waiting time hence maximizes serviceability. In our work, we consider the problem of scheduling dependent tasks (equivalent to PRMs) onto an entire FPGA (equivalent to a single PRR). Our objective is to reduce the makespan (source-to-sink latency). The research

problems and objectives in [8] and in our work are different but complementary. We believe it would be possible (and scientifically interesting) to combine the two works into a more complete DSE engine capable to study the PRR-to-PRM mapping and the PRM scheduling within each PRR.

The work in [9] describes a hypervisor for multi-tenant FPGAs that manages the resources of PRRs and assigns them to hardware tasks. The focus of the work in [9] is on building the hypervisor framework rather than proposing specific algorithms to map and/or schedule hardware tasks. This is evident from Fig. 5 in [9], that illustrates the interactions between the PRR monitor and the Virtual Device Manager: no policy is specified to allocate hardware tasks in users requests. The PRR-to-PRM mapper in [8] as well as our Slot algorithm could be implemented as part of the infrastructure in [9] to take management decisions for PRRs and PRMs.

2.1. Virtual FPGAs

Motivated by the above mentioned needs for multi-tenancy, FPGAs are the subject of many studies on virtualization. The most popular approach that is currently taken to virtualize hardware FPGA resources (both programmable and non-programmable) is based on the concept of overlays. An overlay, or intermediate fabric implements a reconfigurable architecture within the reconfigurable logic of a physical FPGA. This approach can be compared to the Java virtual machine concept that abstracts a physical CPU. An overlay consist of a virtual arrangement of some physical resource or set of physical resources connected by routing channels to a physical interconnect network (e.g., switch grid, crossbar). Overlays offer great portability and flexibility and are of interest, besides cloud computing, as they allow workloads to be ported to FPGAs from different vendors or families. An application vendor can generate and distribute software and configuration bitstreams that can directly execute on different target FPGAs, without the need for further target-specific synthesis or place-and-route steps. In the literature, overlays can be classified as either fine or coarse grained, depending on the granularity of the physical resources that

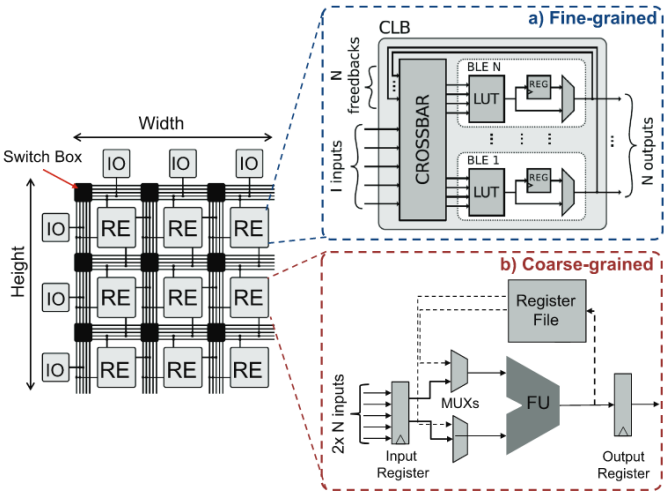


Figure 1: The structure of fine-grained (a) and coarse-grained (b) overlays (taken from [10]).

they abstract. Fig. 1, that we report from [10], shows overlay architectures for physical logic elements. In a fine-grained overlay architecture [11, 12], virtual reconfigurable elements (RE in Fig. 1, left-hand side) denote an arrangement of generic logic elements (denoted Configurable Logic Elements - CLBs, in Fig. 1a) composed of LUTs and hardware registers that receive inputs from a crossbar. In a coarse-grained overlay architecture [13, 14], virtual reconfigurable elements denote (Fig. 1b) an array of coarse-grained physical resources, namely register files and functional units (FU in Fig. 1b) that implement general-purpose operations (e.g., addition, multiplication).

Fig. 2 shows an overview of the development flow for virtualized FPGA applications (workloads). Virtual bitstreams are generated from a process of virtual synthesis that targets a virtual overlay architecture rather than a physical device architecture. In this process, the generic overlay primitives are mapped to a target FPGA. Users are offered with a view of the FPGA as a virtual device that is composed of overlays as well as of a hypervisor that governs the execution of virtual tasks onto the virtual resources.

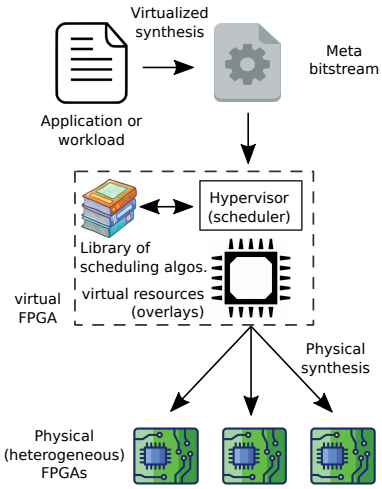


Figure 2: Overview of the development approach for virtualized FPGAs.

2.2. Basic definitions

Before terminating this section we state some basic definitions that are necessary for the reader to avoid misunderstandings with the multiple variants of FPGA scheduling problems that can be found in the literature, as well as to clearly establish the boundaries of our contribution.

By the term *task*, we denote an elementary unit of work, regardless the abstraction level, that is implemented as a combination of the resources offered by a target FPGA. A term frequently employed in the literature on scheduling, that is equivalent to our concept of a task is "job". By the term *scheduling*, we refer to the process of deciding an execution order for a set of dependent tasks. We also employ the same term to denote an execution order itself, depending on the context. A scheduling *makespan* is the difference in time between the moment where the last task (sink) of a workload terminates executing and the moment where the first task (source) started execution. We consider total FPGA reconfigurations only. In the context of partial reconfigurations, where an FPGA is statically divided in regions that are totally reconfigured, our contribution can be used to target individual regions. Our FPGA *scheduling problem* consists in scheduling a workload of tasks that requires at least one total reconfiguration

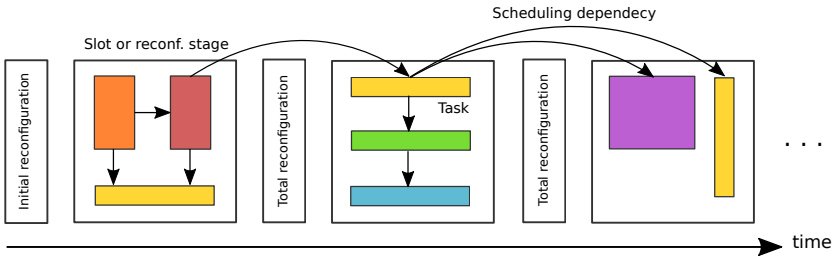


Figure 3: The time diagram of a solution to our FPGA scheduling problem.

before the last task finishes execution (apart from the first initial reconfiguration that occurs just before the source task). The objective of our scheduling problem is to minimize the makespan of a workload of tasks, while satisfying the constraints imposed by resource requirements of tasks. To solve this problem, we propose a heuristic that we call *Slot* as it is based on grouping tasks and allocating them to reconfigurations stages called slots. Two consecutive reconfigurations stages are separated by a total FPGA reconfiguration, Fig. 3. A task allocated to stage n cannot start execution until all tasks from previous stages have terminated. In the rest of this paper, we will use the terms stage and slot (lower case) as synonyms.

An important assumption in our work is that the dependency graphs that we consider denote *single-user* workloads. This removes any concern related to security and to the spatial sharing of the FPGA with multiple, potentially malicious users. Also, please note that because of this assumptions, the serviceability of our scheduling algorithm depends on the algorithm's run-time, solely. Serviceability is defined as the success rate of the allocation of hardware tasks. It is inversely proportional to the average time a user has to wait before its tasks are deployed.

2.3. Target applications

We conclude this section by specifying to the reader the type and characteristics of applications that we target in our work. Here, we focus on applications that are generally called *data-flow* or *stream* applications as they process large

streams of data for image, video and signal processing systems. These applications are composed of tasks interconnected by producer-consumer relations. For this reason, they are typically captured by a dependency graph whose nodes denote tasks and edges denote producer-consumer relations involving the exchange of input/output data between pairs of tasks. The density of these relations is typically low and, based on our experience, it can be quantified around 0.2, on average. We recall to the reader that, in graph theory, the density is a measure of how far the number of edges within a graph is from the maximal number of edges. For directed graphs, the density is defined as $D = \frac{|E|}{|V| \times (|V|-1)}$, where $|E|$ is the number of edges and $|V|$ the number of nodes. The size of these applications, in terms of $|N|$, is typically in the order of magnitude of tens of tasks, depending on the level of abstraction of a design. In the context of our work, we are positioned at system-level of abstraction, where one task corresponds to one block in an application's block diagram or to one function in the application's mathematical algorithm.

In data-flow applications, task priority is somehow implicit in the topology of the dependency graph given by the producer-consumer relations between tasks. In some scenarios, it is possible to enforce the priority of tasks by adding artificial edges that denote control and/or precedence constraints. An example of such a scenario is when periodic applications are considered and it is desired to avoid the overlapping execution of tasks from different periods. In this case, it is necessary to add precedence constraints from tasks in period n to tasks in period $n + 1$.

By means of example, Fig. 4 shows four application dependency graphs from the literature [15] for three image filters (Sobel, SUSAN, RASTA-PLP) and one JPEG encoder. Also, Fig. 5 shows the system configuration of Intel's CTAccel Image Processor architecture [16]. The latter is used to accelerate JPEG, WebP and Lepton decoding and encoding, image resizing and cropped pixel processing. The system targets E-commerce, social network and other domains where it is necessary to accelerate thumbnail generation, image transcoding as well as common image processing functions such as sharpening, watermarking and image

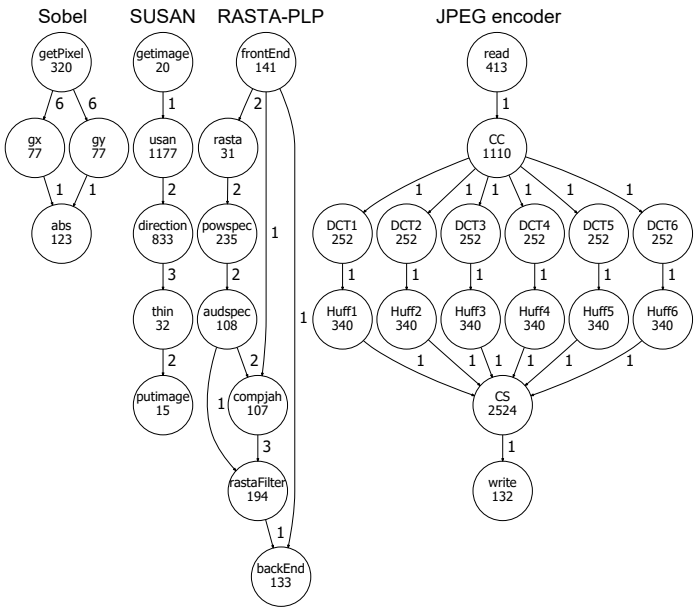


Figure 4: The dependency graphs of example applications for image and video processing from [15]. Nodes are annotated with execution times and edges are annotated with the amount of data exchanged between tasks, according to the Synchronous Data-Flow Model of Computation [17].

analytics. In context of multi-tenant FPGAs, the light-blue blocks in the architecture of Fig. 5 could be mapped to different PRRs, thus enabling concurrent executions of multiple image conversion algorithms for different users/processes. These examples indicate that the average size of the workloads that we target is in the order of magnitude of tens of tasks. This is significantly different from more traditional cloud workloads composed of hundreds of small, independent software tasks (e.g., microservices in telecommunication cloud systems).

3. Related Work

The FPGA scheduling problem that we defined in Section 2 can be seen as a variant of the classical Resource-Constrained Scheduling Problem (RCSP)

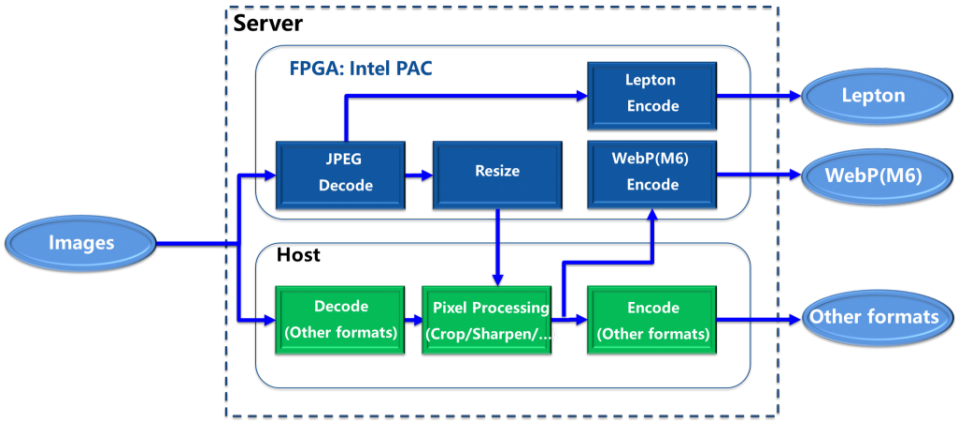


Figure 5: The system configuration of Intel's CTAccel Image Processor architecture, as depicted in [16]. The acronym PAC stands for Programmable Acceleration Card that is an FPGA card with Intel Arrai 10 GX FPGA.

for a project, [18]. A project is composed of T units of work (called activities, jobs, tasks) labeled t_j, $j = 1, ..., J$. Typically two tasks denoted with t_0 and with t_{J+1} represent the start (source) and the end (sink) tasks of the project, respectively. Each task t_j has a duration (processing time) h_j and cannot be preempted. Task t_j may start once its predecessors are finished. The network of precedence relations (dependency graph) is assumed to be acyclic. Tasks request for resources to execute: K resources are given and task t_j requires r_{jk} units of resource k for each execution. The capacity (maximum amount of available units) of resource k is denoted as R_k and is fixed. The goal of a RCSP is to determine a start and finish time for each task such that the project's makespan is minimized. Different variants have been studied in the literature, that differ in the characteristics of tasks and resources (e.g., tasks' execution times and resource requirements can vary, resources can or cannot be renewed to execute a new task). The authors in [19] demonstrated that the classical RCSP belongs to the class of NP-hard problems. Our FPGA scheduling problem differs from the classical RCSP problem because of three fundamental differences:

- At the end of each FPGA configuration stage, no task is running and the FPGA hardware is totally reconfigured to accommodate for the next slot

of task, Fig. 3. On the other hand, in RCSPs, it is possible that, at any given time instant, there is at least one running task.

- The FPGA reconfiguration time cannot be neglected. With respect to the formulation of the classical RCSP, the reconfiguration process can be modeled as an additional task that has a given duration but does not consume resources (idle task). Instead, it must be properly inserted in the global scheduling of tasks, at time t, to regenerate the resources consumed by the tasks scheduled in the slot just before time t. Difficulties arise because the insertion of total reconfigurations impacts the final makespan.

- FPGA resources are shared in both space and time: logic elements, programmable interconnections and I/O blocks are spatially divided in regions that are programmed to execute some tasks; the same region can be assigned to execute different tasks, at different time instants, that require a different configuration of resources. To the best of our knowledge, existing RCSP formulations in the literature concern problems where resources are either space or time shared, but not both.

Because of these differences, the complexity of our FPGA scheduling would deserve its own analysis. Such an analysis is out of scope for this paper: to the best of our knowledge, a proof of equivalence between the RCSP (or any of its variants) and our FPGA scheduling problem does not exist in the literature, yet. Hence, we cannot claim that our FPGA scheduling problem has the same complexity as the classical RCSP. However, our evaluation in Section 5 shows that the run-time of the MILP solver increases very rapidly with an increase in the size of the problem instances. This is a good indication that our problem is hard, thus motivating the need for efficient heuristics.

The two-dimensional sharing of resources mentioned above lead us to the following classification of resources. To the best of our knowledge, we are the first to propose such a classification, that is expected to support how engineers and

researchers consider FPGA scheduling problems, thus advancing the state-of-the-art. This classification is valid for both physical and virtual resources. We classify resources as scheduling-*independent* and scheduling-*dependent*. We define a scheduling-independent resource as one that is exclusively assigned to a task t_j for the entire lifetime of the reconfiguration slot that contains t_j. Consumption of a scheduling-independent resource r_{jk} (e.g., logic element, memory blocks) within a slot can be considered by simply subtracting the desired amount from the capacity of that resource R_k. In other words, the consumption of r_{jk} does not depend on how task t_j is scheduled with respect to the other tasks in a slot (tasks that space-share the remaining resources). Consumption of r_{jk} only depends on the presence/absence of t_j within slot s. We remind to the reader that in our problem, resource and dependency constraints are sufficient conditions to guarantee that tasks, within a slot, execute without conflicts.

On the contrary, a scheduling-dependent resource is one whose consumption depends on how remaining tasks are scheduled within a slot. Examples of scheduling-dependent resources are off-chip memory bandwidth and network bandwidth. Satisfying requests for these resources does not only depend on the capacity for a resource but also depends on the execution order of the remaining tasks in a slot. This difference can be better understood by means of an example. Suppose that three tasks A, B, C are allocated to a slot (regardless the slot allocation policy) as they fit the FPGA scheduling-independent resources (e.g., logic element, memory blocks, DSP blocks). If A and B both require 60% of the off-chip memory bandwidth to manage I/O data and C only requires 30%, it is clear that the correct execution of this slot depends on how memory accesses are scheduled. Thus, with scheduling-dependent resources, resource constraints and dependency constraints are necessary but not sufficient conditions for the tasks within a slot to execute without conflicts.

In the rest of this section, we classify related works based on the quality of the solutions they produce: exact solutions and non-exact solutions (meta-heuristics and heuristics).

3.1. Exact techniques

Exact techniques are based on mathematical formulations such as those underlying Constraint Programming, Integer Linear Programming (ILP) and Mixed Integer Linear Programming (MILP). These formulations are either written by the user or derived from input representations (e.g., code, models) and then fed to a solver (e.g., Microsoft Z3, IBM CPLEX, GNU Linear Programming Kit) together with an objective function (e.g., minimize makespan). The solver can be configured to output all or some of the solutions (e.g., optimal, feasible). These techniques produce optimal scheduling solutions at the price of high run-times (up to hours, days or years according to the size of the problem) that make them applicable to scenarios where scheduling decisions are taken off-line. These run-times are due to the very large solution space which characterizes the FPGA scheduling problem that is influenced by inter-task dependencies, tasks' execution times, tasks' resources, the nature of resources and inter-stage reconfigurations.

Relevant works are those described in [20] that compares a MILP formulation with a list heuristic and a Genetic Algorithm meta-heuristic. In [21], the authors propose an off-line iterative scheduler based on a MILP formulation that allows to consider the impact of latency, peak power, energy consumption or linear combinations thereof. In [22], the ILP formulation targets the minimization of processing time together with the time spent for I/O operations. The authors also consider the execution of multiple instances of a given task. In [23], the authors use an ILP formulation for the knapsack problem. As discussed below in sub-section 3.5, our FPGA scheduling problem is not equivalent to the knapsack problem and requires a different solution. In [24], the authors present a MILP formulation to schedule jobs allocated to networked FPGAs and thus consider the latency impact of send/receive message primitives over the network.

In the context of our work, we also used a MILP formulation (Appendix B) to evaluate the quality of the solutions (distance from the optimum) produced by our heuristic, Section 5.

3.2. Meta-heuristics

Generally speaking, meta-heuristics are procedures used to solve large-scale optimization problems by guiding the search process towards a valid, near-optimal solution. These procedures are based on two general concepts: intensification and diversification. Intensification allows a promising region of solutions to be better explored by improving on the current solution. On the other hand, diversification allows to force the search towards unexplored regions of the solution space, in order to escape from local optima. The most commonly used meta-heuristics for the FPGA scheduling problem are Genetic Algorithms (GAs), Tabu Search (TS), Ant Colony Optimization (ACO) and Simulated Annealing (SA). The run-time, as well as the quality, of meta-heuristic solutions stands in between the long run-times of exact methods and the fast run-time of approximate heuristics. Common run-times for FPGA scheduling meta-heuristics range from one second to a few minutes. The landscape of works based on meta-heuristics is very fragmented: the task scheduling problem is often solved by a meta-heuristic together with other FPGA-related problems (e.g., mapping). These solutions are often applied to solve problems encountered during FPGA synthesis. The authors in [25] demonstrated the superiority of Ant-Colony Optimization with respect to Tabu Search, Genetic Algorithms and Simulated Annealing for the classical FPGA RCSPs. A previous edition of the Elsevier Microprocessors and Microsystems journal [26] presents two algorithms based on ant colony to schedule both dependent and independent tasks onto multiple FPGAs. The problems the authors aim to solve has a twofold objective function: to minimize the energy consumption and to respect a deadline constraint, while accounting for partial reconfigurations taking place in parallel on multiple FPGAs. The authors in [20] propose a genetic algorithm for task scheduling that shows almost linear scalability in terms of the number of generations required to converge. The authors in [27] propose two algorithms based on SA and GA for a target platform composed of a CPU and a FPGA.

3.3. List-based heuristics

List-based heuristics are commonly used to solve the multi-processor scheduling problem that consists in scheduling processors in order to minimize the makespan of a set of partially ordered tasks. List heuristics are common in the RCSP literature because the multi-processor scheduling problem can be formulated as a RCSP and the theoretical complexity of many RCSP variants is based on reductions to multi-processor scheduling problems. In a list-based heuristic, *individual* tasks from a dependency graph are ranked in a priority list and assigned, in sequence, to the earliest available processor that fits their resource request. This process is repeated until all tasks are scheduled. List heuristics require very small run-times (proportional to the size of the dependency graph in terms of nodes and edges) at the cost of a compromise on the quality of output schedules. As it is evident in our evaluation (Section 5), a common characteristic of list-based heuristics is that tasks' priorities are computed by traversing the dependency graph, whose topology strongly influences the quality of output schedules. When adapting list heuristics to FPGAs, a processor is equivalent to a FPGA reconfiguration stage that can physically accommodate for the resources required by a given task. The literature on list heuristics is very large as they are intensively studied since the '70s [28]. As representative examples, we reference here the list heuristics described in [29], called Heterogeneous Earliest Finish Time (HEFT) and Critical Path Of Processor (CPOP). In these algorithms, tasks are assigned to logic processors according to a priority that is computed based on the critical path (in terms of execution time) in a dependency graph. Many variants of HEFT and CPOP were proposed, such as the Next Fit (NF) version. In HEFT, if a task t_j does not fit a logic processor p because of resource constraints, t_j and all higher-rank tasks are assigned to another logic processor. In HEFT-NF, instead, p can execute tasks with higher ranks than t_j, as long as there are available resources. In Section 5, we compare the quality of solutions produced by Slot to those produced by HEFT-NF.

3.4. Group-based heuristics

We classify in this category algorithms that take scheduling decisions for *groups* of tasks. In general, these algorithms require the computation of more complex types of priorities (not only based on a dependency graph's topology) but yield higher-quality schedules. Tasks are grouped together and groups execute sequentially; within a group, tasks can execute in parallel. Typically, a group cannot start before all tasks from previous groups have completed execution. Representative heuristics for FPGA scheduling are described in [4], where groups of tasks are formed based on a task's longest distance from the dependency graph's source and on the tasks' resource consumption. Groups of tasks take different names according to the domain where the algorithms were designed: in real-time systems groups are called servers, in High Performance Computing systems (HPC) groups are called packs or clusters. In compilation, groups of elementary units of work (e.g., functions) are also called clusters.

Numerous contributions (see [30] and its related work) target High-Performance Computing platforms *without* reconfigurable hardware, where tasks are implemented in software on multi-processor CPUs. Scheduling problems in this domain are variants of the classical RCSP or of the multi-processor scheduling problem. While similarities exist between our FPGA scheduling problem and multi-processor scheduling (e.g., both software and hardware tasks consume resources, FPGA reconfigurations can be logically considered similarly to context switches), total reconfigurations have no equivalent in multi-processor systems, where a context switch takes place as soon as a task terminates. To the best of our knowledge, in the HPC domain, only the work in [31] proposes a solution that targets FPGA platforms. Here, resources are abstracted as a single parameter, area. An FPGA area is partitioned in slots, tasks are divided in groups and each group is scheduled to one slot with the Earliest Deadline First policy. With respect to our work, the authors in [31] consider independent tasks and a 1D resource model.

Based on the same rationale as pack scheduling are clustering heuristics that originate in compilers for parallel machines. By means of example, we mention

the work in [32] that is one of the most cited heuristics. Here, groups of tasks called clusters are created, from the dependency graph of some input code, so as to minimize the overall code's execution time. Specific to some solutions in this domain is task duplication: a task may have several copies in different clusters and each copy is scheduled independently. Duplication is not at all a desirable feature in our work: users who rent one or more FPGAs would have to pay also for the resources occupied by duplicate tasks. Another difference is the lower level of granularity that is typical of tasks in clustering algorithms: tasks can be routines or even subparts of routines. To the best of our knowledge, no heuristic exists that forms clusters around high-latency tasks, as we propose in this paper.

Server-based scheduling is a recent technique from the real-time community [33] that groups tasks in so-called servers. It applies to both periodic and aperiodic tasks. It aims to improve the average response time of aperiodic tasks that are scheduled when no instances of periodic tasks are ready to execute. In [33], a server is defined as a periodic task whose purpose is to serve aperiodic requests for resources as soon as possible. It is characterized by a period, a computation time and a set of tasks. To the best of our knowledge, this paradigm has never been studied for task dependency graphs and reconfigurable hardware. Static server-based scheduling for FPGAs was first studied in [34] for independent periodic tasks. On-line server-based scheduling is described in [35, 36], in the context of a real-time operating system, also for independent tasks. The authors propose a heuristic where servers are merged in order to reduce the time utilization factor of a merged server's task set Γ, that is defined as: $U^{(T)} = \sum_{T_i \in \Gamma} \frac{C_i}{P_i}$, where T_i is a task, C_i its computation time and P_i its period. The resulting set of merged servers is then executed sequentially with the single-processor Earliest Deadline First policy.

Our work differs from [34, 36, 35] as (i) we account for task dependencies and (ii) we provide a generic model for K resources whose requests are constant in time and independent of task scheduling.

3.5. Related resource allocation problems

The bin-packing problem (BPP) is a well-known NP-hard combinatorial optimization problem [37]. Different versions of this problem exist. In the original version, items of different volumes v_i must be packed into a finite number of bins, which are in turn characterized by a fixed maximum volume V, in such a way to minimize the number of bins used. Many FPGA scheduling problems are formulated as variants of this problem (e.g., 2D bin packing, where items and bins are rectangles with a given width and height). Here, items are equivalent to tasks: volumes v_i correspond to the resources requested by tasks; a bin's total volume V corresponds to the maximum number of resources offered by the FPGA; the instantiation of a new bin corresponds to the instantiation of a new reconfiguration stage. Despite the analogies, there are some important differences between existing versions of the BPP and our FPGA scheduling problem, which invalidate the re-use of BPP solutions for our problem. Items, in BPP, are associated to consumption of resources only, whereas in our problem, the hardware execution time of tasks must also be considered. Instantiation of a new reconfiguration stage increases the total makespan, whereas instantiating a new bin is costless. The objective function of BPP is to minimize the number of bins. In our problem, instead, the objective function is to minimize an application's makespan, which is not necessarily achieved by minimizing the number of reconfigurations stages. Few high-latency stages may require more run-time than a larger number of low-latency stages, even when accounting for the reconfiguration time of all stages.

Bin-packing-related variants of the FPGA scheduling problem are usually solved for static scenarios where the arrival time of tasks and their characteristics are known in advance. In a previous edition of the Elsevier Microprocessors and Microsystems journal [38], an algorithm is presented for online task scheduling with unknown arrival and execution times, for the case where tasks are assigned to 2D rectangles of FPGA resources. This algorithm solves the problem called Maximal Empty Rectangles that consists in finding an empty rectangle of maximal area within a 2D matrix (that models the FPGA), whose

entries are either empty (unoccupied) or occupied by a task.

The knapsack problem [39] is another well-known combinatorial optimization problem related to resource allocation. In its original version, the user is given a set of different items, each characterized by a weight and a value. The objective is to determine which items to put in a knapsack so as to maximize the value of the selected items, while respecting the size constraints of the knapsack. Our FPGA scheduling problem resembles the knapsack problem, in which the FPGA resource capacities are equivalent to the capacity of the knapsack, items' weights correspond to tasks' resource needs and items' values correspond to tasks' hardware execution time. However, a solution to the classical knapsack problem is not necessarily also a solution to our problem. Scheduling tasks in one reconfiguration stage is equivalent to solving the knapsack problem for one knapsack. Hence, scheduling a set of tasks that cannot all fit the resources of an FPGA, could be seen as the equivalent of solving the knapsack problem on a set of identical knapsacks. However, this equivalence is fallacious as there is no equivalent for the FPGA reconfiguration time in the knapsack problem.

3.6. Summary of differences with related works

To help the reader understand the positioning of our work, we conclude this section by means of a summary of the differences between our heuristic and the heuristics referenced above. In list-based heuristics, tasks are sorted in a list according to a score that is based on the topological position of tasks in the dependency graph. As a result, list-based scheduling is heavily biased by inter-task dependencies. On the contrary, in our heuristic, groups of tasks are formed around dominating tasks which can be located anywhere in the dependency graph. Groups are formed on the basis of a score that accounts for both the topology and the tasks' characteristics (resources, execution time) which are statistically independent variables.

With respect to group-based heuristics, our algorithm is novel as it proposes a new grouping strategy that, to the best of our knowledge, has never been pro-

posed in the literature. With respect to bin-packing problems, our contributions differs in that we aim to reduce the application's makespan rather than the total number of bins. With respect to knapsack-related problems, we account for the FPGA reconfiguration time by attempting to form as few groups as possible in order to limit the number of FPGA reconfigurations.

With respect to the contributions presented in the conference paper [40], we present here a richer evaluation. In [40], the evaluation was conducted on a smaller benchmark, it compared Slot against HEFT-NF and involved a slower implementation of the two algorithms in Java. In this paper, the evaluation results are based on a richer benchmark and on faster, C implementations of the algorithms Slot, HEFT-NF *and* HPF-NF.

4. The Slot heuristic

In this section, we present the Slot heuristic, that we first published in [40]. We start by describing the *logical* modeling of an FPGA and our design assumptions. Because this logical modeling applies to physical or virtual FPGAs, in the rest of the paper, the term FPGA (unless evident from the context) will denote a logical FPGA. We use the term logical to stress the fact that our modeling and scheduling heuristic can be applied to both physical and virtual devices/resources. Subsequently, we present the heuristic's pseudo-code, an instructional example and an algorithm the formulation of Slot to reduce FPGA resource fragmentation. We conclude this section with a discussion on the application of our heuristic to scheduling problems that occur in synthesis and workload management.

4.1. Logical FPGA modeling and design assumptions

Our reference logical platform, as shown in Fig. 6, is composed of a logical static region and a logical reconfigurable region. The static region has a general-purpose processor (e.g., physical FPGA micro-processor, host server CPU) controlling the reconfiguration and a reconfigurator device that internally

reconfigures the system at runtime (e.g., hypervisor in the case of a virtual FPGA). The reconfigurable region is entirely dedicated to execute a workload (denoted User application in Fig. 6) and maps either to a physical or to a virtual FPGA. This assignment is fixed for the entire execution of a workload.

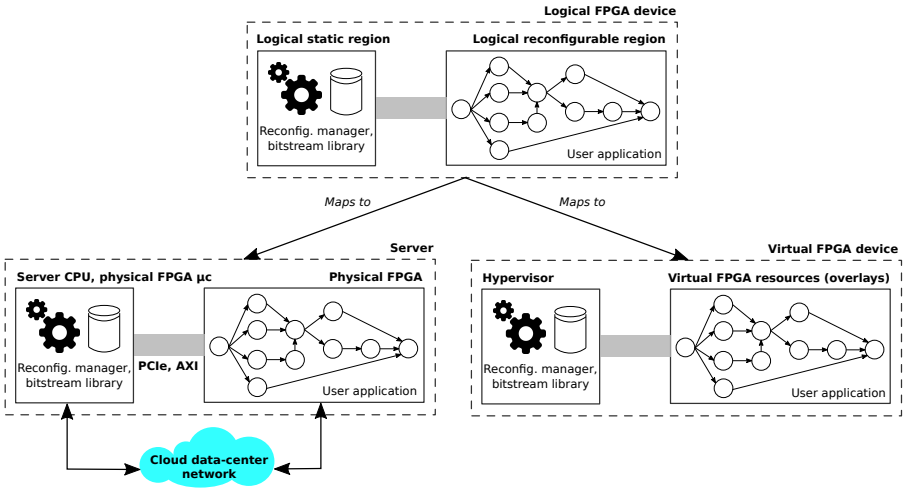


Figure 6: The mapping of a logical FPGA system to a physical device (bottom left sub-figure) and a virtual device (bottom right sub-figure).

In the context of our work, each workload disposes of one or more logical bitstreams that are designed off-line either by the user or are available as part of a library developed by third-parties, e.g., the cloud provider or the FPGA manufacturer. We target the case where a workload cannot execute in its entirety on a given FPGA and the latter must be reconfigured at least once after the workload has started executing. These assumptions are coherent with the design capabilities currently offered by FPGA manufacturers and vendors' tools as well as by virtualization technologies (Section 2).

A workload is denoted, as in Fig. 6, by a directed acyclic graph (DAG) DAG $G =< T, E >$. Each task $t_j \in T$ executes on the reconfigurable region, it is a technologically *mapped* netlist implementing the j^{th} logical task (mapped to either physical or virtual/overlay resources). Task j_0 denotes the source task and t_{J+1} denotes the sink task, with $J = |T|$ being the number of actual

tasks. As it is frequent for dependency graphs, the source and sink can be artificial tasks that correspond to units of work that do not occupy resources and have zero execution time. We characterize task t_j by means of a tuple $(h_j, r_{j1}, r_{j2}, ..., r_{jk})$, where h_j is the hardware execution time (HET) taken by t_j to execute. The reconfigurable resources needed by t_j are expressed by the generic tuple $(r_{j1}, r_{j2}, ..., r_{jk})$. Thus, for a physical device, r_{j1} may represent the number of logic blocks, r_{j2} may represent the amount of on-chip RAM, r_{j3} may denote the number of DSP blocks, etc. When applied to a virtual device, such as the fine-grained overlay in Fig. 1a, r_{j1} may represent the number of CLBs, r_{j2} may denote the number of I/O blocks, etc. Note that, for instance, for easier partial bitstreams composition, the logic block resource could very easily be replaced by entire rows of logic blocks. The occupancy of resources in the tuple is associated to an operating frequency. Multiple tuples for different operating frequencies can be assigned to a workload. Our heuristic supports aperiodic as well as periodic applications. We only require a workload DAG to include: one instance of a periodic application's dependency graph for each period to schedule; the precedence relations between tasks that belong to different periods.

In our work, we consider that users are always aware of the characteristics of the workloads they deploy on a target family of FPGAs. In other words, tasks' resource occupancy is known beforehand, typically thanks to data available during the synthesis and simulation of a workload's bitstream, profiling or interpolation and curve-fitting from historic data. Task DAGs can also be readily retrieved with the same techniques or simple static analysis of application code (e.g., dependency analysis available in compilers and hardware synthesis tools).

Other **design assumptions** are listed below:

- All tasks T in a DAG are released at the same time instant.
- The time to read, write and transfer the input/output data for a task t_j in different memory locations is included in its execution time h_j.
- Tasks require a fixed amount of resources and have a fixed execution time.

- The time to transfer a reconfiguration bitstream is included in the FPGA total reconfiguration time T_R.

We require tasks to be designed without pipelining between a producer and a consumer tasks. In workloads that do not respect this constraint, pipelined tasks must be modeled as a single task in the workload's DAG. These assumptions are driven by the context of cloud data centers, where FPGAs are available for multiple users as a general-purpose reconfigurable platform for different types of workloads.

4.2. The heuristic's pseudo-code

The formulation of our heuristic is generic and valid for k-dimensional models of resources. We consider a set K that contains k resources, each available in R_k units. Each task t_j consumes a fixed amount of each resource, r_{jk}. Our heuristic takes scheduling decisions for groups of tasks that we call a *slot*. A slot s is defined by the tuple $(G_s, h_s, r_{s1}, r_{s2}, ..., r_{sk})$. $G_s \subseteq G, G_s = \langle N_s, E_s \rangle$ is the slot's graph of tasks and h_s is the slot's HET. The generic tuple $(r_{s1}, r_{s2}, ..., r_{sk})$ denotes the slot's occupancy for each of the K resources (physical or virtual). Resources occupied by a slot correspond to the sum of the resources occupied by all its tasks. Obviously, the amount of a slot's resources cannot be larger than those available in the target device: $r_{s1} = \sum_{t_j \in G_s} r_{j1} \leq R_1$, $r_{s2} = \sum_{t_j \in G_s} r_{j2} \leq R_2$, ..., $r_{sk} = \sum_{t_j \in G_s} r_{jk} \leq R_k$. Slots are executed sequentially, tasks within a slot cannot execute until all tasks in preceding slots have terminated. Slots are interposed by FPGA reconfigurations that add a latency denoted by T_R.

In short, our heuristic iteratively transforms a DAG that expresses multiple partial execution orders for tasks into a DAG that expresses a single total execution order (the final schedule) for groups of tasks. This is performed, at each iteration, by creating a slot, based on the concept of *computational dominance*. A slot is built around the task that has the highest HET (dominating task) among unscheduled tasks. Selected dominated tasks are added to a slot, as long as there are enough resources, in a way that avoid reducing the parallelism for further slots. The final schedule is a succession of FPGA configurations, whose

latency is determined by the dominating tasks that hide the latencies of the dominated tasks.

Algorithm 1 shows the heuristic's pseudo-code. Its main loop, lines 5-14, iterates over a worklist where tasks are sorted in decreasing order of their HET. At each iteration, the algorithm selects from the worklist a dominating task t_j and computes the set S of candidate slots, function *buildCandidateSlots*(). A candidate slot is composed of a dominating task t_j and a *resource-feasible* set of dominated tasks. Such a set is composed of all combinations of tasks that can execute in parallel to t_j and fit the remaining resources. For instance, let's consider the DAG in Fig. 7a. Let's suppose that t_3 is the dominating task and that both t_2 and t_5 can be allocated to the same slot. There is a set of 4 candidate slots: $S = \{\ \{t_3, t_2, t_5\},\ \{t_3, t_2\},\ \{t_3, t_5\},\ \{t_3\}\ \}$.

```
1  Function generateSlots( G = < T, E > ):
2      G' := G; /* Copy G to G', G'=<T',E'>                          */
3      worklist ← T' \ {t_0, t_{J+1}}; /* Remove source and sink tasks */
4      worklist ← sortInDecreasingOrderOfHET(worklist);
5      foreach t_j ∈ worklist do
6          S ← ∅;/* set of candidate slots                           */
7          S ← buildCandidateSlots(t_j, G', S, R_1, R_2, ..., R_k); /* Defined in
               Algorithm 4                                           */
8          foreach s ∈ S do
9              scores[ s ] ← computeScore(s, G'); /* Defined in Algorithm 2
                   */
10         end
11         G_s ← retrieveLowestScoreSlot(scores[]);
12         G' ← contractSubgraph(G_s, G');
13         worklist ← worklist \ {N_s};
14     end
15 return G';
```

Algorithm 1: The Slot scheduling heuristic

Among all candidate slots in S, only one is selected to be created in the current DAG G', lines 8-11 in Algorithm 1. This selection is based on the score returned by function $computeScore(s, G')$, Algorithm 2. The rationale underlying the score is to compute (an estimate of) the makespan in the residual graph $G' - G_s$, that would result if we created slot s and removed its tasks G_s from G'. This (estimated) makespan is computed by separately considering the impact

1 **Function** `computeScore(` *slot* s, *dependency graph* G' `):`
2 $Y := G' - G_s$;`/* subtracting` G_s `from` G' `*/`
3 $\bar{r}_{Y1} := \frac{\sum_{t_j \in Y} r_{j1}}{R_1}$; $\bar{r}_{Y2} := \frac{\sum_{t_j \in Y} r_{j2}}{R_2}$; ...; $\bar{r}_{Yk} := \frac{\sum_{t_j \in Y} r_{jk}}{R_k}$;
4 $n_Y^{reconfig} := max(\lceil \bar{r}_{Y1} \rceil, \lceil \bar{r}_{Y2} \rceil, ..., \lceil \bar{r}_{Yk} \rceil)$;
5 **return** $T_\infty(G' - G_s) + n_Y^{reconfig} \times T_R$;

Algorithm 2: The function that assigns a score to a slot.

of the variables that constrain our scheduling problem. These variables are the inter-task dependencies and HETs (accounted by the first term at line 5 in Algorithm 2); the reconfiguration time T_R and the occupancy of tasks' resources (accounted by the second term at line 5 in Algorithm 2). We separately consider the impact of these variables, by means of two terms (line 5 in Algorithm 2) as they are statistically independent. The first term, $T_\infty(G' - G_s)$ quantifies the impact of tasks' HET and inter-task dependencies, by ignoring resource occupancy. We compute it as the sum of the HETs, T_∞ for all tasks that lie on the critical path from source to sink in the subgraph $G' - G_s$. This term is similar to the score computed by list-based heuristics (e.g., HEFT in Section 3). To enhance the precision of such a score, we consider a second term that is an estimate of the number of reconfigurations, in the residual graph $G' - G_s$. It is computed by ignoring inter-task dependencies and considering the occupancy of the tasks' resources only. It is denoted as $n_Y^{reconfig}$ in Algorithm 2, where $Y = G' - G_s$.

Back to Algorithm 1, at line 11, we select the slot with the lowest score. This is the slot for which the estimated makespan in $G' - G_s$ is the lowest. Therefore, creating this slot leaves the (estimated) highest degree of parallelism in the

residual DAG $G' - G_s$. Creating a slot is performed by contracting the nodes for the slot's tasks G_s into a single node, in G', by function *contractSubgraph*().
650 The latter modifies G' by relabeling nodes that belong to G_s with the new slot identifier. It collapses nodes in G_s by removing edges internal to G_s as well as duplicate cross edges (edges with an endpoint in G_s and one in $G' - G_s$) and self-loops (edges whose endpoints are both in G_s). For instance, the contraction of tasks t_2, t_3, t_5 in Fig. 7a results in the DAG on Fig. 7b.

655 We precise to the reader that, in Algorithm 2, function $computeScore(s, G')$ does not modify G'. Instead, function $contractSubgraph(G_s, G')$ returns a modified version of G' that, at line 12 in Algorithm 1, is used for the next iteration and overwrites the current G'.

660 With respect to the classification of resources as scheduling dependent and independent, in Section 3, the current formulation of Slot efficiently considers scheduling-independent resources. This is because, the constraints imposed by inter-task dependencies and scheduling-independent resources are sufficient to establish a total execution order for all tasks in a slot. This does not hold when
665 the task model includes scheduling-dependent resources. As discussed in Section 3, let's suppose that three tasks A, B, C are allocated to a slot as they fit the scheduling-independent resources. Let's also suppose that A and B both require 60% of the off-chip memory-bandwidth to access I/O data (scheduling-dependent resource), while C only requires 30%. Regardless the inter-task de-
670 pendencies, the constraints imposed by the memory-bandwidth needs are not sufficient to define a total execution order for A, B, C. It is thus necessary to select a total execution order by means of some additional policy that decides how memory accesses are scheduled. An example of such a policy is an algorithm that schedules memory accesses based on the bandwidth required by each task,
675 so that the memory bandwidth consumed by all scheduled tasks is maximized. Nevertheless, in Slot, scheduling-dependent resources can be treated as if they were scheduling-independent at the price of a more pessimistic output schedule.

4.3. Example

We illustrate our heuristic on the example DAG in Fig. 7a (continued in Fig. 8). We apply our contribution to a physical FPGA device, namely the Xilinx Virtex Ultrascale 9P [41], which is available on servers of Amazon Elastic Compute Cloud. This FPGA disposes of 2586k logic elements, 6840 DSP blocks, 75.9 Mb of embedded memory blocks and a reconfiguration time of 200 ms (for a bitstream of 76.45 MB and a SPIx4 bus at 100 MHz). We remind to the reader that the availability of target resources in a specific device does not influence our heuristic, as the capacity and the consumption of resources can be normalized and expressed in the interval $[0, 1]$.

To illustrate how Slot solves our FPGA scheduling problem for multiple resources, we consider three types of resources: logic elements r_{j1} (LEs, called Configurable Logic Blocks, CLBs, in Xilinx nomenclature), DSPs r_{j2} and embedded memory blocks r_{j3} (these are on-chip RAM blocks that we call EMBs; they are denoted BRAM in Xilinx' nomenclature). In a more refined example, we could also model, for each task, the number and bitwidth of inputs and outputs which need to be connected to the static part and the communication channels between tasks. Because the application of our heuristic to an instructional example would be difficult to follow with such a modeling, we decided to use a simpler 3-resource model that does not capture these low-level resources.

The resource occupancy of tasks in Fig. 7a are given in Table 1. Fig. 7 and Fig. 8 illustrate all the graph transformations that the heuristic performs from a partially ordered DAG of tasks (Fig. 7a) to a totally order DAG of slots (Fig. 8g). Each transformation corresponds to an iteration of the for-loop in Algorithm 1.

We highlight to the reader the usefulness of the score in Algorithm 2: it allows the heuristic to take scheduling decisions that a user normally considers counter-intuitive. For instance, slot $S_0 = \{t_2, t_3, t_5\}$ is preferred over 19 other candidate slots, such as $\{t_2, t_3, t_4\}$. When S_0 is created, it leaves the highest degree of parallelism for the next graph transformation. While in $\{t_2, t_3, t_4\}$, all tasks can execute in parallel, slot $\{t_2, t_3, t_5\}$ amortizes the execution time of t_5

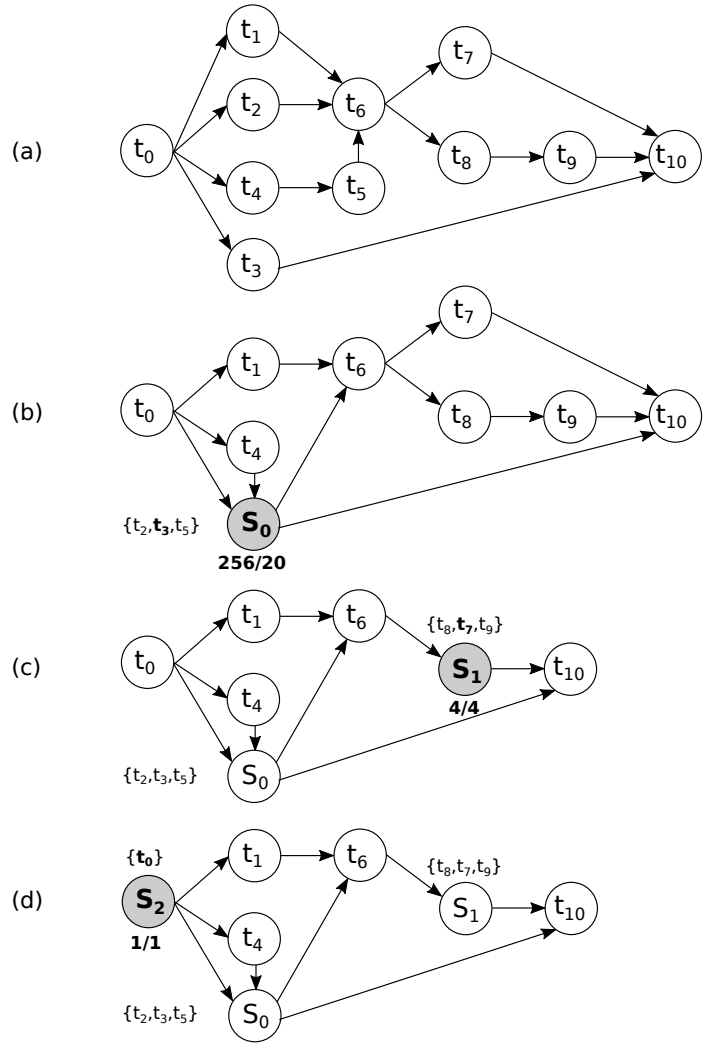
(larger than that of t_4) as t_5 can execute in parallel to t_3 (dominating task).

As indicated at line 5 in Algorithm 2, the score is $T_\infty(Y) + n_Y^{reconfig} \times T_R$, where $Y = G - G_s$, G is the dependency graph in Fig. 7a, $G_s = \langle N_s, \emptyset \rangle$ is the subgraph formed by tasks $\{t_2, t_3, t_5\}$, only (no edges between tasks $\{t_2, t_3, t_5\}$ exist in the dependency graph in Fig. 7a). The term $T_\infty(Y)$ is given by: $h_{t_0} + max(h_{t_4}, h_{t_1}) + h_{t_6} + max(h_{t_7}, h_{t_8+t_9}) + h_{t_{10}} = 287 + 200 + 199 + 303 + 114\ [ms] = 1103\ [ms]$ (we remind to the reader that h_{t_j} denotes the hardware execution time of t_j). In the second term, that indicates the minimum number of reconfigurations, $n_Y^{reconfig} := max(\lceil \bar{r}_{LEs} \rceil, \lceil \bar{r}_{DSPs} \rceil, \lceil \bar{r}_{EMBs} \rceil)$, where $\lceil \bar{r}_{LEs} \rceil = \lceil 5090k/2586k \rceil = 2$, $\lceil \bar{r}_{DSPs} \rceil = \lceil 10266/6840 \rceil = 2$, $\lceil \bar{r}_{EMBs} \rceil = \lceil 81/75.9 \rceil = 2$. Hence, $n_J^{reconfig} = 2$, the second term $n_Y^{reconfig} \times T_R = 2 \times 200\ [ms]$. Thus, the final score is $1103 + 2 \times 200\ [ms] = 1503\ [ms]$.

In Fig. 8, all tasks within slots execute in parallel but for slot S_1, where t_7 executes in parallel to the sequence of t_8, t_9. The total makespan of the slot DAG in Fig. 8g is 1763 ms: it is equal to the sum of the dominating tasks' HET, plus the latency necessary for 5 reconfigurations.

4.4. Reducing the fragmentation of FPGA resources

As illustrated in Fig. 7 and Fig. 8, we compute a schedule by progressively transforming an initial tasks DAG, which defines a partial order for tasks, Fig. 7a, into a slot DAG that specifies a total execution order for both slots and tasks, Fig. 8g. While designing the heuristic, we observed that, in most cases, during the final iterations of Algorithm 1, slots tend to be composed of a single dominating task, see Fig. 8e, Fig. 8f and Fig. 8g. This is because most of the candidate dominated tasks have already been assigned to slots in previous iterations. Thus, the fragmentation of FPGA resources in these single-task slots is very high. It can be reduced by compacting single-task slots and has the indirect benefit of reducing the slot DAG's makespan because it also removes some inter-slot reconfigurations. Multiple approaches exist to reduce the fragmentation of FPGA resources. Based on our experience, we propose Algorithm 3. Here, we scan all slots in the slot DAG and, for each single-task



Legend:
S_n = slot created at iteration n in Algorithm 1
$\{t_0,\ldots,\mathbf{t_k},\ldots\}$ = list of tasks for a slot; $\mathbf{t_k}$ in bold is the dominating task
f/g = complexity encountered in the creation of slot: f (theoretical), g (actual)

Figure 7: The creation of slots in our heuristic on the dependency DAG (a), part I.

Task	**LEs [u]**	**DSPs [u]**	**EMBs [Mb]**	HET **[ms]**
t_0	487k	1966	7	287
t_1	402k	2565	7	139
t_2	272k	1539	15	209
t_3	353k	1966	12	460
t_4	609k	513	18	200
t_5	704k	428	15	314
t_6	943k	1453	10	199
t_7	788k	1881	5	303
t_8	566k	428	5	35
t_9	1004k	599	9	49
t_{10}	291k	855	20	114

Table 1: The resource occupancy and hardware execution time (HET) of the tasks in Fig. 7 and Fig. 8.

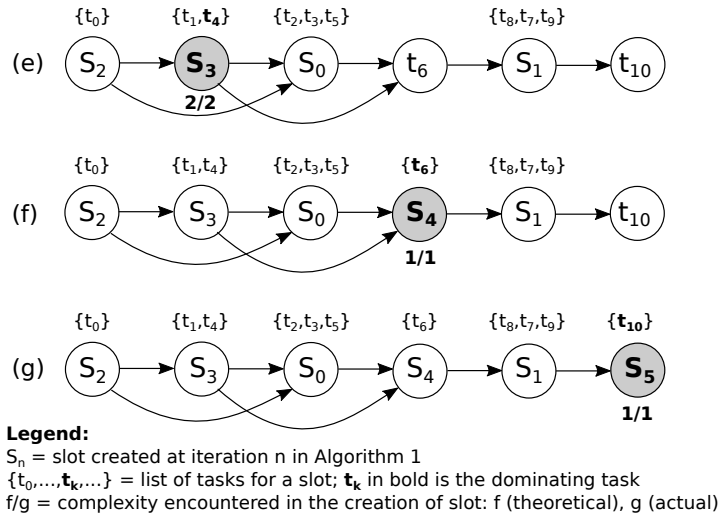


Figure 8: The creation of slots in our heuristic on the dependency DAG (a), part II (continued from Fig. 7.

slot, we attempt to allocate each of its tasks t_j to a neighboring slot, in a first-fit manner. This re-allocation is performed by means of contracting edges between slots. Edge contraction is defined in [42] as the operation that removes an edge from a graph, while merging the edge's end vertices and removing duplicate edges. A task t_j is allocated to the first neighboring slot s' that has enough FPGA resources and for which dependencies are respected. All tasks in s' must either be predecessors or successors of t_j in the initial DAG G. This approach is simple yet efficient enough to produce solutions that are very close to the optimum (see Section 5). Its complexity is $O(|S|)$, that is linear with the number of nodes in the slot DAG.

```
1  G' ← generateSlots(G); /* Defined in Algorithm 1                    */
2  G' ← reduceReconfigurations(G'); /* Pseudo-code below               */
3
4  Function reduceReconfigurations( slot DAG G' =< S, L > ):
       /* S := set of slots, L := set of slot arcs                     */
5      foreach s ∈ S | |G_s| == 1 do
6          foreach s' ∈ {S \ s} | ∀t_i ∈ T_s', t_i ∈ pred(s, G') ∨ t_i ∈ succ(s, G') do
7              if (r_s'1 + r_s1 < R_1) ∧ (r_s'2 + r_s2 < R_2) ∧ ... ∧ (r_s'k + r_sk < R_k) then
8                  contractEdge(s → s', G');
9                  break;
10             end
11         end
12     end
13 return;
```

Algorithm 3: Merging single-task slots in first-fit.

Fig. 9 illustrates how function $reduceReconfigurations()$ reduces the latency for the slot DAG of Fig. 8g. It merges S_2 and S_3 in the new slot $S_{2,3}$ and it merges S_0 and S_4 in the new slot $S_{0,4}$. The improved DAG in Fig. 9c contains 4 slots (instead of 6 in Fig. 8g) and requires only 3 reconfigurations (as opposed to 5 in Fig. 8g). The final makespan is reduced by 22.23%: from 1763

ms in Fig. 8g to 1537 ms (this also coincides with the optimal makespan). This corresponds to 3 times the FPGA reconfiguration time plus the makespans of: the sequence of t_0, t_4 (slot $S_{2,3}$ - latency of t_1 is hidden); the sequence of t_2, t_6 (slot $S_{0,4}$ - latency of t_3 and t_5 is hidden); the processing time of t_7 (slot S_1 - latency of t_8 and t_9 is hidden); the processing time of t_{10} (slot S_{10}).

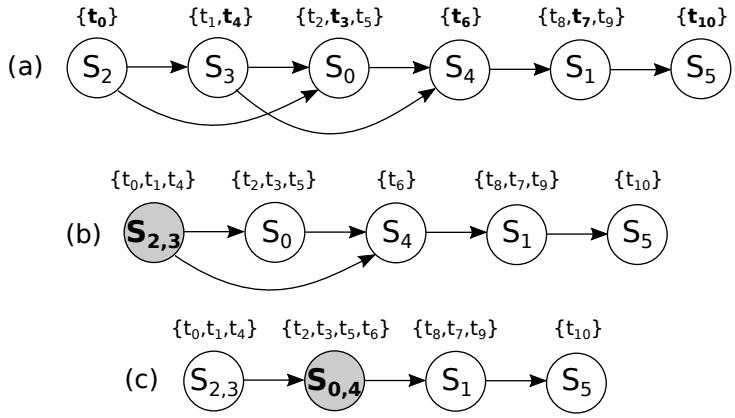


Figure 9: Reducing the makespan of Fig. 8g as described in Algorithm 3.

4.5. The heuristic's complexity

The complexity of the heuristic is dominated by the creation of all candidate slots for a dominating task t_i, function $buildCandidateSlots()$ (line 7 in Algorithm 1) whose pseudo-code is presented in Algorithm 4. Candidate slots are computed from C: a subgraph of the current DAG G', where the dominating task t_i, its successors and predecessors are removed. Function $combinationsOfParallelTasks()$, line 3 in Algorithm 4, returns the c-combinations of tasks in the subgraph $K \subseteq G'$, with $c = 1, ...|C|$ that can be executed in parallel to a dominating task. In Fig. 7a, for the dominating task t_3, this function returns the combinations of $c = 1, 2, ..., 8$ tasks that can execute in parallel to t_3, from the subgraph obtained by removing t_0, t_3 and t_{10} in Fig. 7a.

However, some of these combinations are invalid and must be filtered out (lines 8-13 in Algorithm 4). Invalid combinations contain tasks that do not fit the available resources or violate the computational dominance principle.

```
1  Function buildCandidateSlots( t_j, G' = < T', E' >, S, R_1, R_2, ..., R_k):
2      C ← G' \ {t_j, pred(t_j, G'), succ(t_j, G')};
3      foreach C' ∈ combinationsOfParallelTasks(C) do
4          G_s = {t_j};
5          s ← ({t_j}, h_j, r_j1, r_j2, ..., r_jk);
6          feasible = true;
7          foreach t_c ∈ C' do
8              if executionTime(G_s + t_c, G') ≤ h_i then
9                  if feasibleAllocation(s, t_c, R_1, R_2, ..., R_k) then
10                     s ← (G_s + t_c, executionTime(G_s + t_c, G'), r_s1 + r_c1, r_s2 +
                         r_c2, ..., r_sk + r_ck);
11                     continue;
12                 end
13             end
14             feasible = false;
15         end
16         if feasible == true then
17             S ← S ∪ {s};
18         end
19     end
20 return S
```

Algorithm 4: The function that builds the candidate slots.

Function $executionTime(X, G')$, line 8, is used to verify if a combination of tasks X respects the computational dominance principle in the DAG G'. It returns the length of the critical path that tasks in X form in G'. For $X = \{t_2, t_4, t_5\}$ in Fig. 7a, the function returns $max\{h_2, (h_4{+}h_5)\}$. Function $feasibleAllocation()$, line 9, verifies if a slot disposes of enough FPGA resources for a new task.

For the sake of precision, we specify that functions $pred(t_j, G')$ and $succ(t_j, G')$, line 1, return the set of predecessors (from the source) and successors (up to the sink) of a task $t_j \in G'$, respectively. At line 2, the operator $\setminus$ deletes a set of nodes N from a graph G'. It returns the subgraph $C' \subseteq G'$ that results from removing all nodes in N and all edges incident to nodes in N. Operation $G_s + t_c$, at line 10, adds t_c to the slot task graph G_s. This addition produces the same graph as the subtraction $G - G_s - t_c$.

The complexity of the heuristic is determined by the number of combinations of tasks that may form a slot, for the subgraph C defined at line 2 in Algorithm 4. This number depends on the task dependencies in C and cannot be expressed in closed form. In the worst case, for a graph C where all tasks can execute in parallel, the number of combinations amounts to $\sum_{i=1}^{|N_C|} \binom{|N_C|}{i}$, where $|N_C|$ is the number of tasks in C. This also corresponds to the theoretic case of a graph with no edges (null graph). We ignore this case as it violates our design assumptions: by definition, the input of our heuristic is non-null dependency graph. In fact, the total number of combinations is strongly limited by task dependencies and by resource constraints. For instance, let's consider the DAG in Fig. 7a and the dominating task t_3. Combinations $\{t_4, t_6\}, \{t_4, t_7\}, \{t_4, t_8\}, \{t_4, t_9\}$ are not valid candidate slots (even if they fit the available resources) because t_6, t_7, t_8, t_9 must be scheduled after t_5, which in turn must be scheduled after t_4.

In most of the practical cases we encountered, the complexity is maximal at the first iterations of the loop at line 3 in Algorithm 4. Complexity decreases significantly with the creation of subsequent slots as parallelism in G' is progressively reduced. This can be seen in Fig. 8 where below each slot we reported a pair of numbers f/g. f is the number of combinations in C that can

be computed without considering for inter-task dependencies (the theoretical complexity). g is the number of *valid* candidate slots (the actual complexity). A significant difference between f and g exists only for S_0.

In our implementation, we combined function $combinationsOfParalleTasks()$ with the tests at lines 7 and 10. When a combination of tasks X does not respect the computational dominance condition or requires more FPGA resources than those available, we stop exploring combinations that are descendants of X. This prunes the candidate space and significantly reduces runtime.

Although the complexity of Algorithm 4 is theoretically exponential, the evaluation in Section 5 shows an actual complexity that tends to be polynomial in terms of the size of the input task DAG.

For the sake of completeness, we report in Table 2 the complexity of the functions that compose our heuristic. Apart from Algorithm 4, the functions that contribute to the heuristic's complexity are: the main loop of Algorithm 1 (lines 5-14), function $computeScore()$ in Algorithm 2 and function $minimizeReconfigurations()$ in Algorithm 3. The main loop of Algorithm 1 iterates over a worklist composed of the sequence of tasks that compose the input DAG ordered in terms of decreasing hardware execution time. Thus, iterations at lines 5-14 in Algorithm 1 are executed proportionally to the number of tasks in the dependency graph, in the worst-case. This corresponds to the situation where slots are composed by a single task. In our implementation of Algorithm 2, we computed the score by

Algorithm	**Theoretical complexity**
Main loop of Alg. 1	$O(\|N\|)$
Score - Alg. 2	$O(\|N\| + \|E\|)$
Optimization - Alg. 3	$O(\|slots\|)$
Slot construction - Alg. 4	$\sum_{i=1}^{\|N_C\|} \binom{\|N_C\|}{i}$

Table 2: Computational complexity of the functions in the Slot heuristic.

means of one breadth-first visit of the dependency graph. Thus, the worst-case complexity of Algorithm 2 is $O(|E| + |N|)$. Complexity of Algorithm 3 is linear in terms of the number of slots as it is based on a simple scan of the list of slots.

4.6. A discussion on the application of the Slot heuristic

It is common for works that target FPGA scheduling problems (Section 3) to consider an FPGA as a general-purpose computation unit where hardware tasks can be programmed and un-programmed on-the-fly, similarly to how software tasks are scheduled in a multi-processor system. Nevertheless, the adoption of this approach is hampered by the fact that FPGA tasks are synthesized digital circuits whose execution requires the configuration of heterogeneous arrangements of logic elements, programmable interconnects, I/O blocks and other types of complex resources (e.g., DSP blocks, RAM blocks). Works exist, such as [43] published at Euromicro DSD 2012, that aim to enable the support for the Multiple Input Multiple Data computation model for FPGAs by proposing a run-time architecture composed of a mesh network of reconfigurable modules similar to DSP processors. Unfortunately this type of approach is not mainstream. In the case of FPGAs that support coarse-grain partial reconfigurations (e.g., Xilinx XC6200), an application of our heuristic for dynamic task scheduling is possible. This was successfully demonstrated (with another heuristic, of course) in the '90s, as described in [44]. This type of FPGAs (now outdated) were effectively composed of homogeneous logic that could be combined together to implement hardware tasks. However, modern FPGAs offer heterogeneous, fine-grain resources that make it much more difficult, for design tools, to directly support dynamic scheduling. The reason for this is that such a dynamic scheduling requires hardware support for the relocation of tasks to different regions of physical resources. A more practical application of Slot for dynamic scheduling is to floorplan a target FPGA and generate bitstreams for all tasks at all possible reconfigurable regions and store the bitstreams in a data-base, similarly to the approach in [45]. In general, the application of fast heuristics to dynamic scheduling is hampered by the long runtime of the place-

and-route phase that are necessary to produce the hardware configuration for the scheduled tasks. For this reason, more precise yet time-consuming scheduling approaches (e.g., based on exact methods or meta-heuristics) may be more appealing than heuristic-based approaches.

In the context of dynamic scheduling, Slot also applies to dynamic workloads, i.e., dependency graphs that change over time due to the arrival of new tasks to be scheduled. Here, the slots must be generated (Algorithm 4) on the dependency graph updated with new edges and nodes corresponding to the arrival of new tasks. The Slot algorithm can be applied to dynamic workloads in scenarios where the algorithm's run-time is smaller than the reconfiguration time T_R.

Another application opportunity for Slot is provided by works that analyze and translate or manipulate bitstreams [46]. These works aim to (re)generate and (re)parameterize on-the-fly new bitstreams (e.g., reroute wires, reconfigure LUTs and clock signals) from existing designs, without re-running the entire design flow. This allows to rapidly program similar reconfigurable regions to execute different tasks, thus reducing the costs of dynamic scheduling. In the 2010 edition of Euromicro DSD, the work in [47] demonstrated the feasibility of creating partial FPGA reconfigurations at run-time. The applicability of these approaches, however, is still hampered by the high time required to produce bitstreams (47.5 seconds on average, on a Xilinx Virtex-II Pro and a 300 MHz Power PC processor, as reported in [47]) and by tool support. To counter this latter issue, in Euromicro DSD 2016, a tool called AutoReloc, based on Xilinx PlanAhead tool, was proposed to automate floorplanning and timing constraints management [48].

Virtualization approaches for FPGAs, such as those based on overlays [10], are an additional opportunity for dynamic scheduling based on Slot. As described in Section 2, overlays allow to decouple logical designs from the characteristics of physical devices similarly to the virtualization offered by the Java Virtual Machine for software tasks. Although this type of approaches are still in their infancy, our heuristic could be used in the future, by the hypervisor of

such virtual FPGAs, to dynamically schedule virtual tasks.

Apart from dynamic scheduling, another application of our work is for the partitioning of tasks into reconfigurable regions. This partitioning is a combinatorial optimization problem that is solved off-line and consists in deciding the size and number of reconfigurable regions that will execute hardware tasks at run-time. This partitioning phase is currently not supported by vendor tools. Solutions to this problem are the result of compromises between conflicting requirements in terms of resource usage and reconfiguration time. For instance, one solution could be designed to combine tasks into few large regions. While this allows to optimize the resource occupancy, it increases the reconfiguration time that is directly proportional to the amount of resources to be reconfigured (size of regions). Assigning tasks to few large regions also usually entails that these regions are reconfigured more often. This partitioning problem cannot be solved efficiently without considering the scheduling of tasks onto regions. In this context, the makespans of execution orders, produced by Slot, for different sizes of a target region can help designers in improving the quality of solutions to the partitioning problem.

The scheduling problem is also present in High-Level Synthesis (HLS) flows. HLS is a design process that "interprets" the behavioral description of an algorithm, written in a high-level programming language (e.g., C/C++), and produces the digital hardware that implements that behavior. HLS is a commonly supported by FPGA vendors (e.g., Xilinx Vivado HLS, Intel HLS Compiler) as it significantly eases the programming of reconfigurable devices. Scheduling is at the heart of HLS: it determines in which clock cycle an operation will occur by taking into account user directives for control and data flows. HLS scheduling is, of course, constrained by the FPGA resources and our heuristic can be applied in a way similar to how clustering heuristics schedule operation in programming language compilers (sub-section 3.4). Slot can be used, as part of a HLS compiler's back-end, to group low-level operations (e.g., additions, multiplications) specified in a control-flow dependency graph that is extracted by the HLS compiler's front-end from the input code.

5. Evaluation

In this section we report our evaluation of the Slot heuristic in terms of the quality of solutions and of the computational complexity. To compare the quality of solutions with respect to the optimum and to concurrent heuristics we relied on a MILP formulation whose details are presented in Appendix B. To conduct our evaluation, we designed a random generator of FPGA scheduling problem instances, described in Appendix A, with which we generated two benchmarks: one with realistic DAGs (that is, with constrained number of edges), and another with totally random DAGs. The need for this random graph generator arised as, in the context of our target applications (subsection 2.3) there's no publicly available graph generator that allows to tune the topology of candidate graphs (e.g., density of edges, number of input and output edges per node). Additionally, there's no public available benchmark that offers a statistically significant collection of data-flow workloads for pertinent evaluations. The constraints we added to our graph generator to obtain realistic DAGs are based on our experience of real-world designs for signal and image processing systems. We start this section by presenting the evaluation results on the first benchmark of realistic DAGs, subsequently we describe the observed computational complexity and compare solutions produced by Slot with those produced by concurrent heuristics. Finally, we also present the evaluation results on the second benchmark of totally random DAGs.

5.1. Evaluation results

The target *physical* FPGA we used for the evaluation is the Xilinx XC7S25 from the Spartan-7 family. We selected a different device from the one in the instructional example in Section 4 to further prove that performance of our contribution do not depend on the target device. We modeled its resources in terms of logic elements, RAM blocks and DSP blocks. The XC7S25 FPGA is a relatively small FPGA but the complexity of our FPGA scheduling problem does not depend on the size of the target FPGA. Indeed, resource requirements

could as well be normalized and represented as real numbers in the interval $[0, 1]$. The results of our evaluation were retrieved with:

- A MILP solver written in C using the GNU Linear Programming Kit (GLPK) [49] version 5.0 and based on the MILP formulation presented in Appendix B. As the MILP-solving can take much longer than heuristics, a time-out of 24 hours has been set: for any instance exceeding this real time the solver has been stopped, and the instance has been excluded from the comparison with MILP. These "hard" instances have however been retained for comparisons not involving MILP.

- The Slot, HEFT-NF and HPF-NF heuristics have been implemented in C. Slot makes use of version 0.8.5 of the igraph library [50].

HEFT-NF is the Next-Fit version of the list-based heuristic called Heterogeneous Earliest Finish Time [29] discussed in Section 3. It is the variant of the original HEFT algorithm suitable when targeting reconfigurable devices. HPF-NF (High-Priority First, Next Fit) is a group-based heuristic presented in [4]. HPF-NF first sorts tasks in a priority list based on a task's distance from the DAG source and on the task's resource consumption. The priority list is then traversed and tasks are grouped in the same reconfiguration stage if there are enough available resources and if, for a task t_j, all of its immediate predecessors have been scheduled in the same or in a previous group.

We selected these two heuristic as they are representative for the category of list and group based approaches; they are easy to implement and can be readily adapted to our problem without biasing the comparison of scheduling solutions. Comparisons to other heuristics is not possible without significantly denaturing them and biasing the comparison. Indeed, some of the works cited in Section 3 are based on single resource models that are still valid for less recent FPGAs (without embedded DSP or on-chip RAM blocks). Other related heuristics are based on design assumptions that conflict ours. For instance, the contribution in [51] is based on partial reconfiguration; in [31], independent tasks are grouped, whereas we account for dependent tasks.

All experiments were run on a workstation with 2 sockets, 16 hyper-threaded cores per socket, that is, 64 logical CPUs, clocked at 3.5 GHz and with 64 GB of memory. Due to the very large set of instances up to 60 runs have been launched in parallel. The memory monitoring showed that the memory was not a bottleneck in terms of performance. For each problem instance the three heuristics each produced one approximate solution. The schedule, the makespan and the tool's CPU User Time (CUT) have been recorded in a database. All instances with between 6 and 15 tasks have been solved by MILP in less than 24 hours. We did not continue with larger instances for two reasons: First, the total CUT spent on the $n_t = 15$[1] tasks batch already represents about 7 days of cumulated CUT[2]. Second, the preliminary results with larger instances show that the 24 hours time-out starts being exceeded and, unsurprisingly, that the portion of instances for which the time-out is exceeded increases with the number of tasks, which probably introduces a significant bias by keeping only "simple" instances. Continuing with larger instances would take huge computation times and produce more and more biased batches.

5.2. The practical complexity of the FPGA scheduling problem

One of the first outcomes of our experiments is the practical complexity of the FPGA scheduling problem. On the 7500 solved instances we estimated the practical complexity of the FPGA scheduling problem from the CUT taken by our MILP solver. The CUTs were all measured with the `getrusage` function of the `glibc` library. As the complexity increases with the number of tasks, we analyzed the distribution of the CUTs for batches of problem instances with identical number of tasks. Table 3 summarizes the observations for $6 \leq n_t \leq 15$, one row per batch, with the number of instances solved by MILP

[1] In the rest of this section, by n_t we denote the total number of tasks in a dependency graph: $n_t = J + 2$. We previously denoted by J the number of actual tasks (excluding source and sink).

[2] This is without counting the fact that the real time, that we did not record, is always significantly larger than the CUT

(Solved), the total CUT spent on the solved instances (Total), the minimum, maximum, average, standard deviation and median of the CUT. All CUT are in microseconds. Zero values are due to the limited resolution of `getrusage`. The CUT distributions of MILP solving are highly biased towards the minimum

Tasks	Number of instances	Total	Min	Max	Mean	Std	Median
6	300	8.87E+05	0.00E+00	9.75E+03	2.96E+03	1.62E+03	3.05E+03
7	400	3.66E+06	0.00E+00	3.04E+04	9.15E+03	4.27E+03	8.53E+03
8	500	1.37E+07	2.63E+03	7.36E+04	2.75E+04	1.26E+04	2.56E+04
9	600	4.58E+07	1.07E+04	3.47E+05	7.63E+04	4.47E+04	6.60E+04
10	700	1.72E+08	2.70E+04	1.80E+06	2.46E+05	1.75E+05	2.02E+05
11	800	6.49E+08	7.90E+04	8.77E+06	8.11E+05	6.95E+05	6.62E+05
12	900	2.49E+09	2.09E+05	2.55E+07	2.77E+06	2.64E+06	2.02E+06
13	1000	1.10E+10	5.41E+05	2.21E+08	1.10E+07	1.91E+07	6.38E+06
14	1100	1.01E+11	1.11E+06	4.17E+09	9.21E+07	3.17E+08	2.54E+07
15	1200	4.59E+11	3.44E+06	4.24E+09	3.82E+08	6.53E+08	1.31E+08

Table 3: MILP CPU user times in micro-seconds

as can be seen for the $n_t = 15$ tasks batch (1200 instances) and its histogram represented on Figure 10. There are more than 6 orders of magnitude between the non-zero minimum (1.2 milliseconds) and maximum (1.2 hours), and more than 95% of the instances fall in the first decile (0 to 7 minutes). Because of the

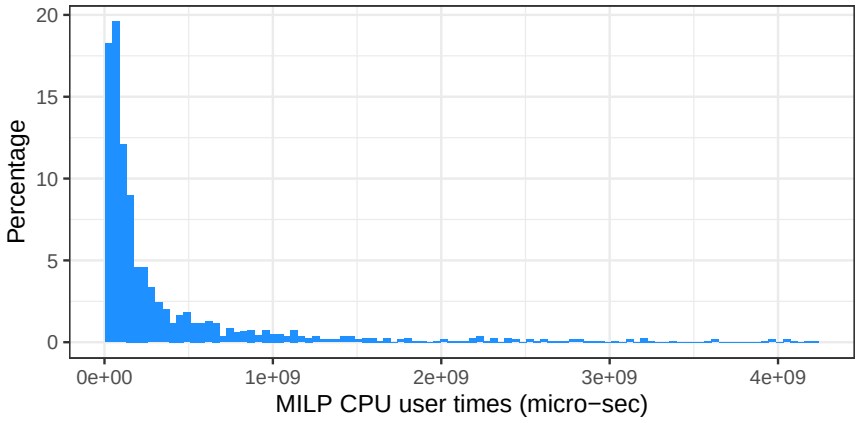


Figure 10: Histogram of MILP CPU user times for $n_t = 15$ tasks

great variability of the CUT and the bias in each batch towards the minimum, we decided to use the median instead of the average to compare the different batches. They are plotted in Figure 11 with a logarithmic scale. Even if these

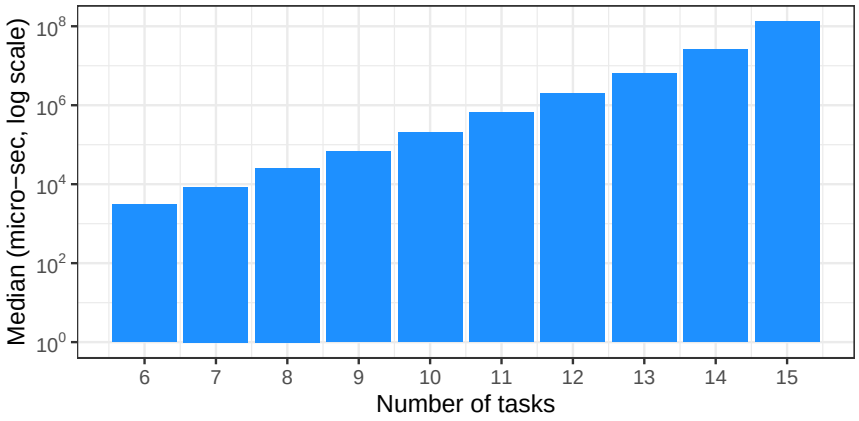


Figure 11: Median of MILP CPU user times vs. the number of tasks

results are not conclusive on the complexity of our FPGA scheduling problem, we clearly see that, in practice, its complexity increases very rapidly with the size of the problem instances.

5.3. Comparison of Slot, HEFT-NF and HPF-NF quality

For the 7500 instances solved with the MILP approach, the criteria we retained to compare the quality of the schedules computed by the heuristics is the additional makespan they add to an optimum solution.

The Empirical Cumulative Distribution Function (ECDF) is commonly used to represent the distribution of quality metrics when comparing optimization techniques. We thus chose to plot the ECDF of the heuristics' over-makespans, expressed as percentages of the optimal makespan. Let us denote M_{ref}, M_s, M_{he}, M_{hp} the optimal makespan, the makespans of solutions by *Slot*, HEFT-NF and HPF-NF, respectively. These makespans, considered as random variables, define the over-makespans Δ_s, Δ_{he} and Δ_{hp} as follows, where $x \in \{s, he, hp\}$

for *Slot*, HEFT-NF and HPF-NF:

$$\Delta_x = 100 \times \frac{M_x - M_{\text{ref}}}{M_{\text{ref}}}$$

And we plot:

$$\begin{aligned} \text{ECDF}_x : \mathbb{R}^+ &\to [0,1] \\ \delta &\mapsto P\left(\Delta_x \leq \delta\right) \end{aligned}$$

The plots of the ECDF_s, ECDF_{he} and ECDF_{hp} for all task batches are shown on Figure 12. These ECDF curves clearly show that *Slot* outperforms both

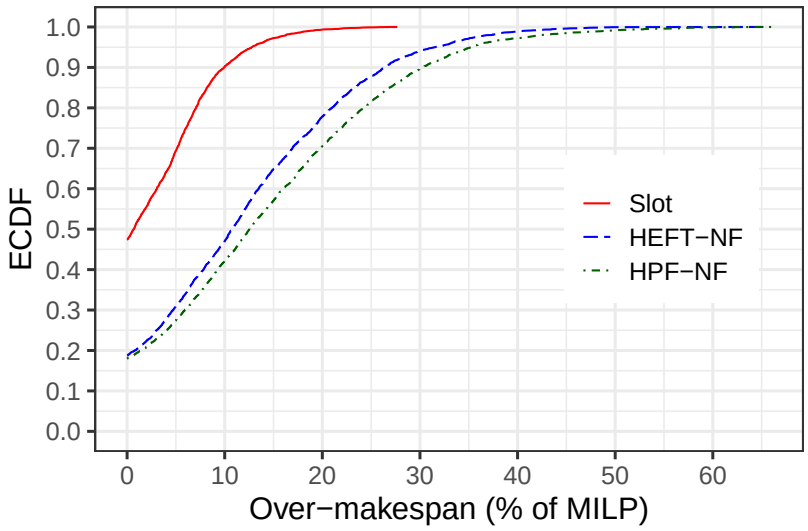


Figure 12: ECDF of *Slot*, HEFT-NF and HPF-NF over-makespans vs. MILP, all batches

HEFT-NF and HPF-NF. Indeed, *Slot* over-makespan is less than 10% on more than 90.1% of the cases against 47.0% for HEFT-NF and 42.1% for HPF-NF. In order to analyze how this advantage over HEFT-NF and HPF-NF depends on the number of tasks we also plotted separately on Figure 13 the individual ECDF curves per batch. The plots show that the *Slot* behavior is significantly better for all explored numbers of tasks, with a considerable advantage to *Slot* for small numbers of tasks. They also show that the quality of the heuristics decreases when the number of tasks increases, which is not very surprising for a problem

whose complexity seems to be exponential. To compare the performance of

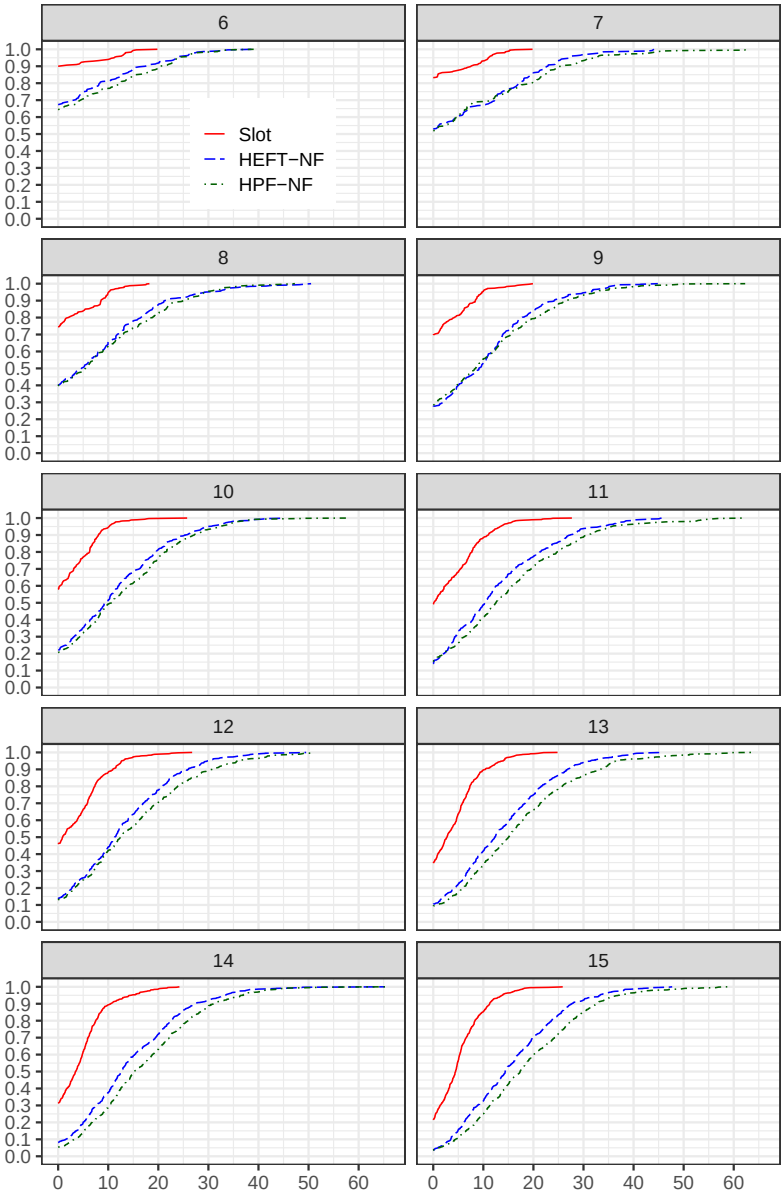


Figure 13: ECDF *Slot*, HEFT-NF and HPF-NF over-makespans vs. MILP, separate batches

Slot, HEFT-NF and HPF-NF on the instances for which the MILP solver has

not been used we do not have an absolute optimum reference anymore. We thus imagined a new heuristic, BEST, that simply consists in taking the best of the heuristics' solutions. The ECDF curves of the over-makespans of *Slot*, HEFT-NF and HPF-NF vs. BEST for all batches are shown on Figure 14. The

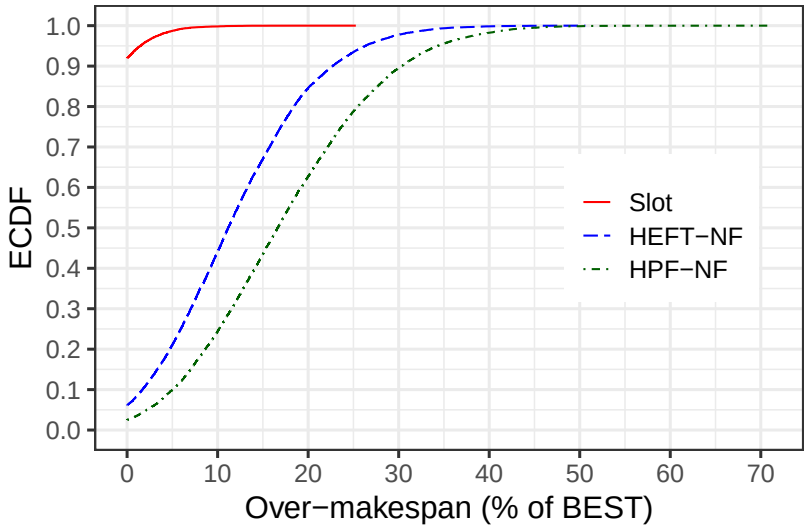


Figure 14: ECDF of *Slot*, HEFT-NF and HPF-NF over-makespans vs. BEST, all batches

individual ECDF curves per $16 \leq n_t \leq 30$ batch are shown on Figure 15. Again, these ECDF curves show that *Slot* outperforms HEFT-NF and HPF-NF in most cases.

5.4. Comparison of the heuristics CPU user times

We did our best to implement MILP, *Slot*, HEFT-NF and HPF-NF as efficiently as possible. MILP CPU user times have already been discussed in Section 5.2. Tables 4, 5 and 6 summarize the observed CUT for the three heuristics on a subset of the batches. Figure 16 shows the median of the CUT for the 3 heuristics as a function of the number of tasks.

Just like MILP, *Slot* has a fast growing complexity with the number of tasks. However, the median CUT for the 15 tasks instances is 5 milliseconds, that is, several orders of magnitude less than with MILP (1.31E+05 milliseconds).

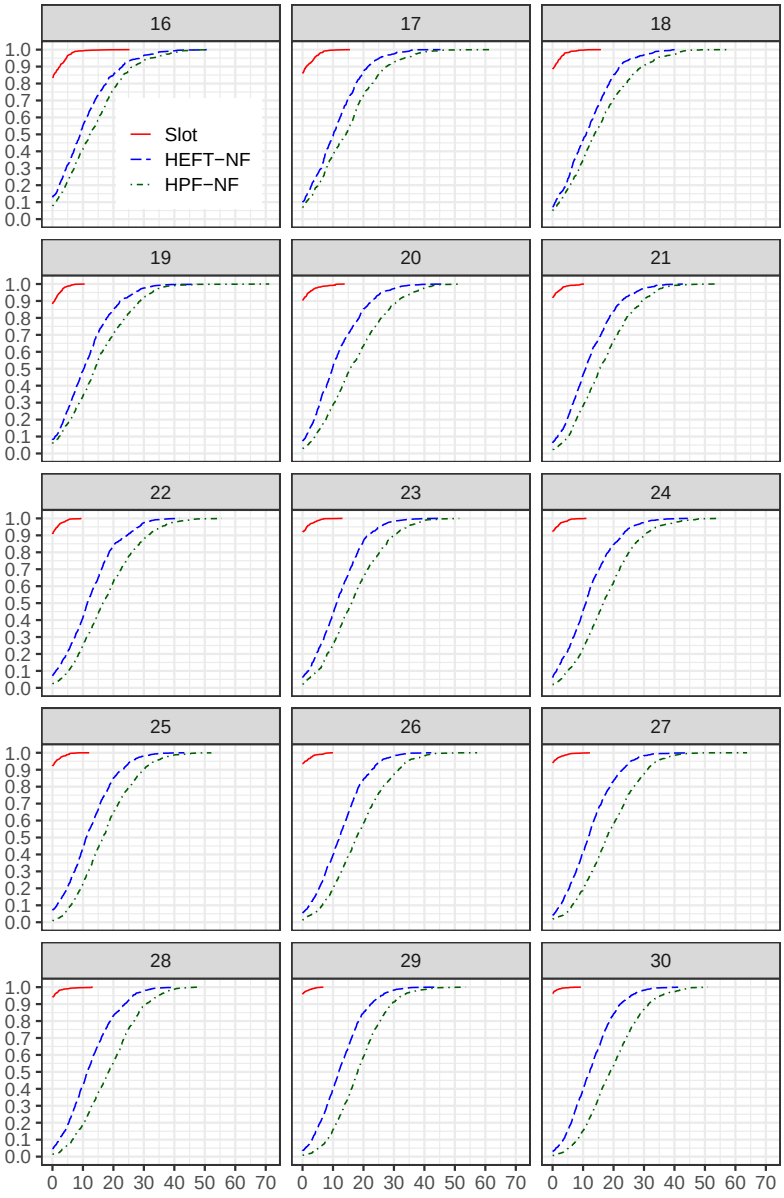


Figure 15: ECDF *Slot*, HEFT-NF and HPF-NF over-makespans vs. BEST, separate batches

Tasks	Number of instances	Total	Min	Max	Mean	Std	Median
10	700	7.30E+05	0.00E+00	2.94E+03	1.04E+03	4.88E+02	1.08E+03
15	1200	6.05E+06	0.00E+00	1.73E+04	5.04E+03	2.27E+03	4.53E+03
20	1700	2.65E+07	3.62E+03	4.48E+04	1.56E+04	6.52E+03	1.49E+04
25	2200	6.81E+07	4.95E+03	1.09E+05	3.10E+04	1.37E+04	2.86E+04
30	2700	1.62E+08	1.50E+04	2.93E+05	6.01E+04	3.33E+04	5.16E+04

Table 4: *Slot* CPU user times in micro-seconds

Tasks	Number of instances	Total	Min	Max	Mean	Std	Median
10	700	6.56E+03	0.00E+00	3.00E+01	9.37E+00	6.12E+00	1.00E+01
15	1200	1.51E+04	0.00E+00	6.10E+01	1.26E+01	8.00E+00	1.30E+01
20	1700	2.73E+04	0.00E+00	8.30E+01	1.61E+01	1.08E+01	1.70E+01
25	2200	4.22E+04	0.00E+00	8.10E+01	1.92E+01	1.22E+01	1.90E+01
30	2700	5.95E+04	0.00E+00	1.16E+02	2.20E+01	1.44E+01	2.20E+01

Table 5: HEFT-NF CPU user times in micro-seconds

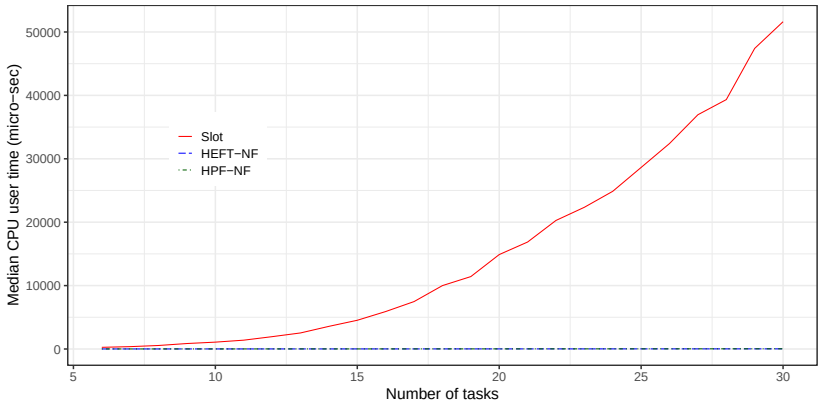


Figure 16: Median CUT of heuristics vs. number of tasks

Even for the most complex instances with 30 tasks the *Slot* CUT is about 1 second. Regardless the quality of our implementations, these CUT also show that HEFT-NF and HPF-NF scale much better than *Slot* with the increase of the number of tasks. As expected the CUT for HEFT-NF and HPF-NF seems linear, while the *Slot* CUT grows much faster with the number of tasks. For very large instances the *Slot* advantages in terms of quality of the computed

Tasks	Number of instances	Total	Min	Max	Mean	Std	Median
10	700	6.98E+03	0.00E+00	5.80E+01	9.97E+00	6.79E+00	1.00E+01
15	1200	1.63E+04	0.00E+00	5.50E+01	1.36E+01	8.94E+00	1.40E+01
20	1700	3.29E+04	0.00E+00	9.80E+01	1.94E+01	1.20E+01	2.00E+01
25	2200	5.64E+04	0.00E+00	8.80E+01	2.56E+01	1.58E+01	2.60E+01
30	2700	8.18E+04	0.00E+00	1.16E+02	3.03E+01	1.88E+01	3.35E+01

Table 6: HPF-NF CPU user times in micro-seconds

schedules will be counterbalanced by its larger run-times.

Before concluding this sub-section, we report the CPU run-time and the quality of solutions for a workload of 100 tasks, in Fig. 17 and Table 7, respectively. These results correspond to a workload that is beyond the size of the reasonable workload we target in our work (i.e., tens of tasks). Nevertheless, we believe these results are a relevant indication to readers interested in the scalability of our contribution for larger workloads, for instance, in the context of resource-constrained scheduling of software tasks on non-reconfigurable hardware platforms.

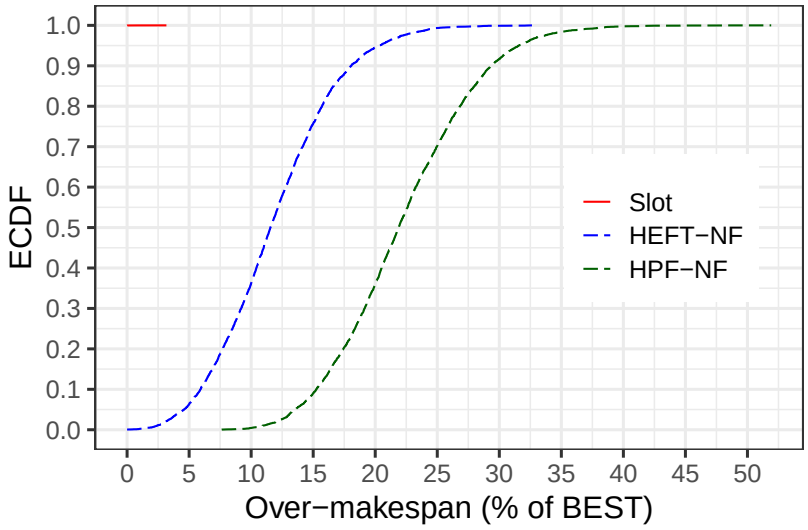


Figure 17: ECDF of *Slot*, HEFT-NF and HPF-NF over-makespans vs. BEST, **100 tasks**

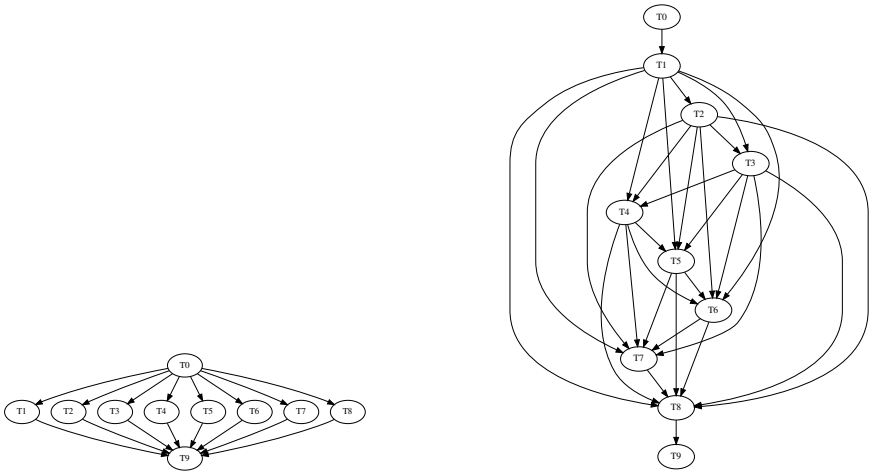
Solver	Number of instances	Total	Min	Max	Mean	Std	Median
HEFT-NF	9700	7.15E+05	0.00E+00	2.28E+02	7.37E+01	3.91E+01	7.20E+01
HPF-NF	9700	4.47E+06	0.00E+00	1.28E+03	4.61E+02	2.53E+02	4.33E+02
Slot	9700	1.29E+12	7.29E+06	1.23E+09	1.33E+08	1.19E+08	9.92E+07

Table 7: CPU user times in micro-seconds for **100 tasks** workloads

5.5. Problem instances with completely random DAG

As already noted the constraints we added to our graph generator to obtain realistic DAGs are based on our experience of real-world designs for signal and image processing systems. Because DAGs from other domains may have different edge densities, in this section, we evaluate our contribution on problem instances with more random graph topologies. The vertexes can now have an arbitrary large number of incident edges. For each number of tasks $6 \leq n_t \leq 30$ and for each possible number of internal edges $0 \leq n_e \leq \frac{(n_t-3)\times(n_t-2)}{2}$ we generated totally random instances. As the number of $n_t \times n_e$ combinations is very large we generated 100 instances per combination where $6 \leq n_t \leq 15$ and only 10 per combination where $15 \leq n_t \leq 30$.

As with the first benchmark, in the $6 \leq n_t \leq 15$ batches we tried to obtain an optimum solution by MILP. Several instances exceeded the 24 hours time-out but as it was only a small number we decided to keep the corresponding batches. We just eliminated from comparisons with MILP the 29/37000 instances for which the time-out was exceeded. This second benchmark contains 70050 instances with completely random DAG, among which 36971 have an MILP optimal solution. The number of internal edges range from 0 to 378 and the total number of edges range from 5 to 380. Figure 18 shows two DAG (with source and sink tasks) of the 10 tasks batch, one with no internal edges and the other with the maximum: 28. The plots of the ECDF_s, ECDF_{he} and ECDF_{hp} for all batches with a known MILP optimal solution are shown on Figure 19. The individual ECDF curves per batch are plotted separately on Figure 20. The plots show that the *Slot* improvement over HEFT-NF and HPF-NF still exists for all explored numbers of tasks but it is less significant than in the first



(a) Zero internal edges

(b) 28 internal edges

Figure 18: Randomly generated 10 tasks DAG

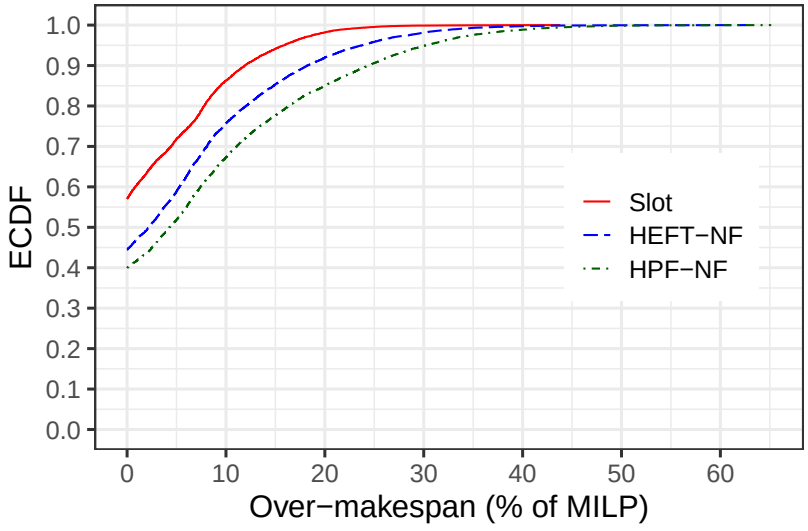


Figure 19: ECDF of *Slot*, HEFT-NF and HPF-NF over-makespans vs. MILP, all batches

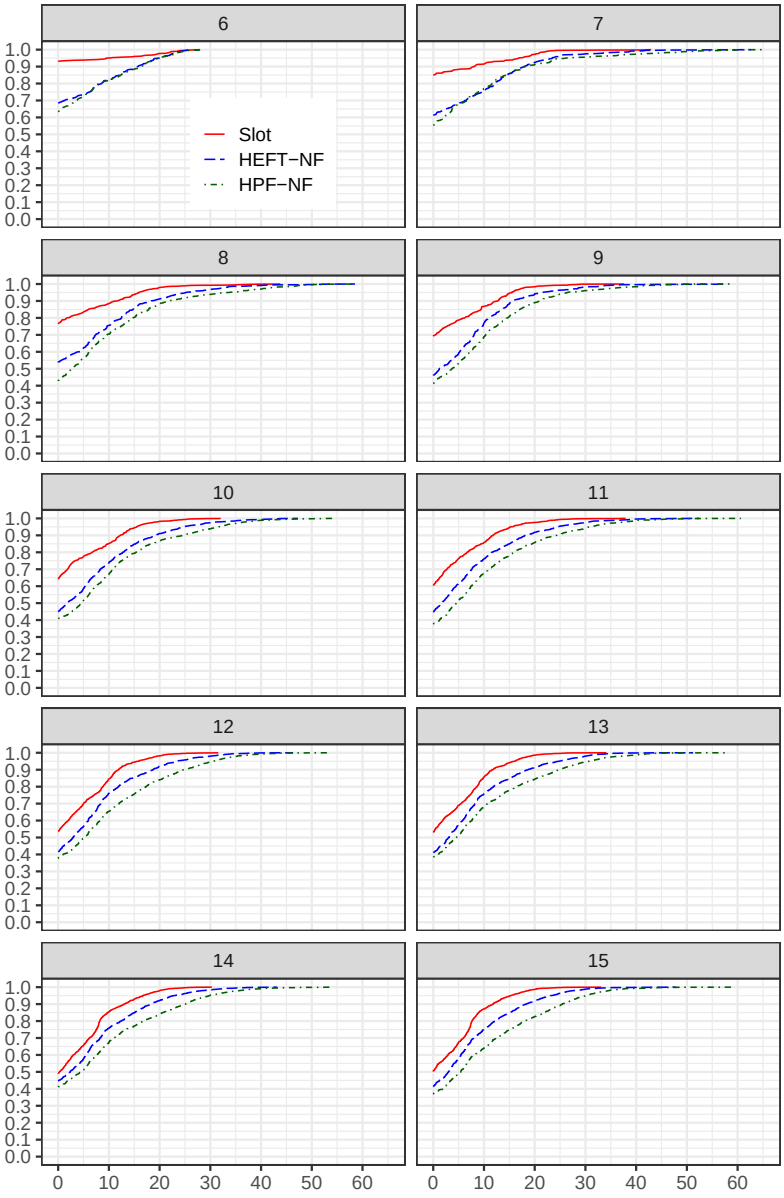


Figure 20: ECDF *Slot*, HEFT-NF and HPF-NF over-makespans vs. MILP, separate batches

benchmark.

The performance of *Slot*, HEFT-NF and HPF-NF on the 16 to 30 tasks instances for which the MILP solver could not be used are compared thanks to the same BEST reference heuristic we already used with the first benchmark. The ECDF curves of the over-makespans of *Slot*, HEFT-NF and HPF-NF vs. BEST for all batches are shown on Figure 21. The individual ECDF curves per

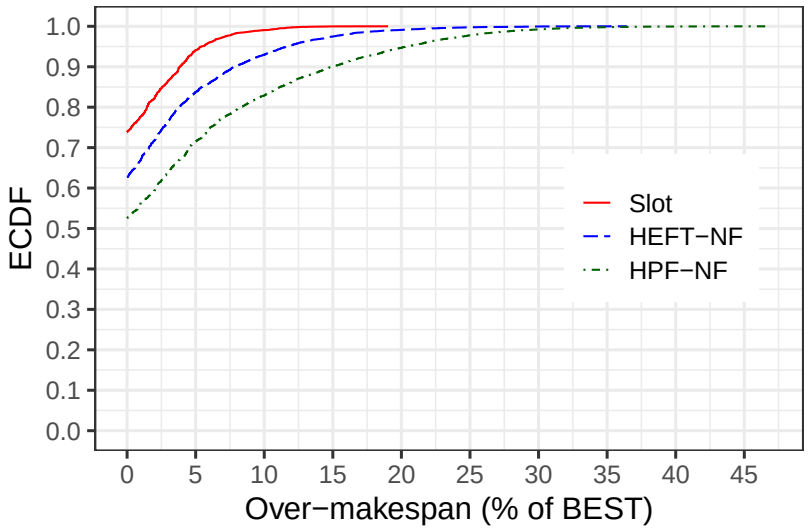


Figure 21: ECDF of *Slot*, HEFT-NF and HPF-NF over-makespans vs. BEST, all batches

$16 \leq n_t \leq 30$ batch are shown on Figure 22. Again, the *Slot* advantage over HEFT-NF and HPF-NF is visible, though it is not as evident as in the first benchmark.

6. Conclusions and Future Works

In this paper we presented a heuristic, that we call Slot, for the scheduling of dependent tasks (i.e., logical units of work) on FPGAs, subject to constraints imposed by the resource requirements of tasks. Our heuristic can be classified as a group heuristic as it consists in forming groups of tasks that are assigned to reconfiguration stages, called slots. Two consecutive slots are interposed by

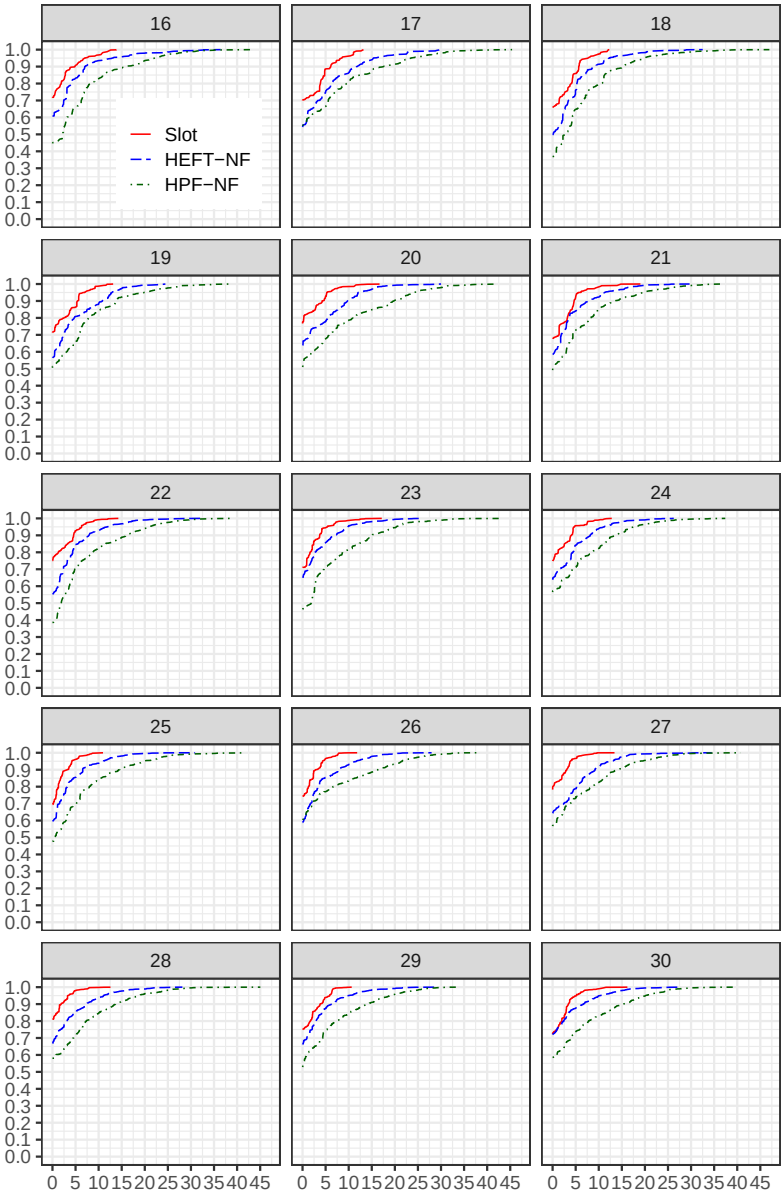


Figure 22: ECDF *Slot*, HEFT-NF and HPF-NF over-makespans vs. BEST, separate batches

total reconfigurations; a task from slot n cannot start executing until all tasks from slot $n-1$ terminate. Our algorithm is based on a parameterized and generic modeling of FPGA resources that allows to capture any resource and to target both physical and virtual FPGAs.

An interesting direction for future work is to study the scheduling when tasks are modeled to consume what we defined as scheduling-dependent resources. These are resources (e.g., network and/or memory bandwidth) whose consumption depends on the scheduling of tasks within a reconfiguration stage. Another interesting direction is to study the case where multiple identical instances of a task are present in a dependency graph. This scenario was studied in a previous publication of the Elsevier Microprocessors and Microsystems journal, [52], for multimedia systems. This scenario is of interest for cloud computing systems, where image and video processing applications are among the most common type of applications being offloaded to reconfigurable hardware.

Acknowledgements

This work was supported by Nokia Bell Labs France as part of an academic partnership with Telecom Paris on Models and Platforms for Network Configuration and Reprogrammability.

Appendix A. Random generator of FPGA scheduling problem instances

An instance of our FPGA scheduling problem is characterized by a workload of tasks and their characteristics (e.g., hardware execution time, resource occupancy), the task dependency graph and a target device. We decided to randomly generate task characteristics, within certain reasonable bounds. More in details, we configured our generator to discard tasks with small resource requirements as these tasks diminish the impact of the resource constraints in favor of the impact of the execution time constraint, on a problem's instance. This would bias the problem space towards that of a classical multi-processor scheduling problem, e.g., we could have problem instances where all workload tasks fit in a single slot because tasks consume few resources. We also prevented the generation of tasks with large resource requirements as this corresponds to simple problem instances where solutions tend to be formed by single-task slots. Consequently, we decided that, given a random problem instance, each task requires a proportion of a resource comprised between 10% and 50% of the total capacity for that resource. In terms of our formalization (Section 4): $0.1 \times R_k \leq r_{jk} \leq 0.5 \times R_k,\ \forall j = 1, 2, ...J,\ \forall k = 1, 2, ..., K$. Our experiments show that this choice produces many "hard" problem instances with non-trivial solutions.

A similar reasoning was conducted to establish the generation of the tasks' hardware execution time (HET). Short HETs tend to bias the problem towards a resource-only combinatorial optimization problem, while long HETs tend to favor our heuristic, based on dominant tasks first. Thus, we decided that a task's HET is comprised between 1/4 and 4 times the FPGA reconfiguration time, $0.25 \times T_R \leq h_j \leq 4 \times T_R,\ \forall j = 1, 2, ..., J$. The dependency graph generation is a bit more subtle because it must be acyclic, have a unique source and sink nodes and because of the need to produce "realistic" graphs, in terms of density of edges, that can represent real-world workloads. Based on our experience on the design of signal and image processing systems, real-world dependency graphs are

not very dense: tasks with more than 3 producers and/or consumers are rare. Thus, our random graph generator is defined by the following characteristics:

- The graph random generator is given a number of vertices (tasks) $J \geq 1$ and a number of internal edges (dependencies) n_e. An internal edge is an edge between any of the J actual tasks, thus an edge whose starting and arrival nodes are both different from the source vertex t_0 and the sink vertex t_{J+1}. We remind to the reader that J denotes the number of actual tasks, excluding the source and the sink tasks.

- The task indexes are used as a topological ordering of the dependency graph: if there is an edge from t_i to t_j, then $i < j$ in the topological ordering. This is a necessary and sufficient condition to guarantee that the graph is acyclic.

- The generator starts with a graph with $J + 2$ vertices and no edges. It iterates n_e times and adds one internal edge per iteration. At each iteration the generator randomly selects two different vertices t_i and t_j such that $0 < i < j < J + 1$ and there is no edge already placed between them. In order to avoid unrealistic numbers of incident edges the two vertices must also be such that after adding an edge between them:

  - Their total number of input internal edges is less or equal 2.
  - Their total number of output internal edges is less or equal 4.
  - Their total number of incident internal edges is less or equal 5.

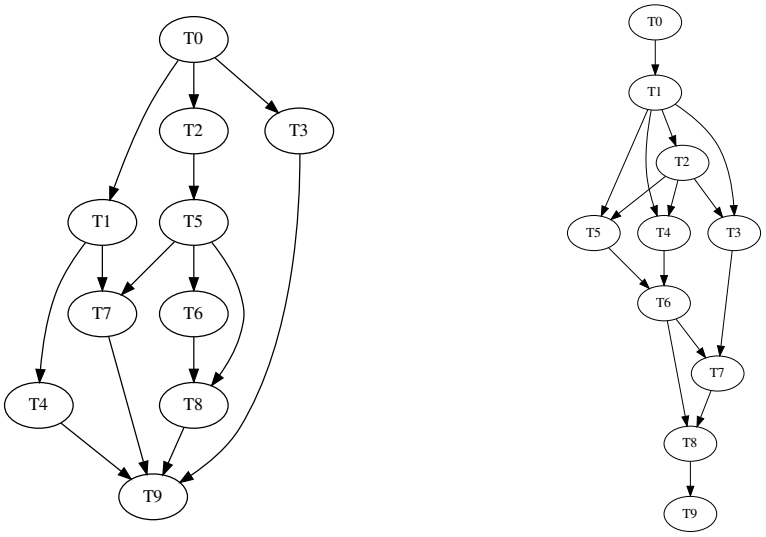
- The generator then adds an internal edge from t_i to t_j. At the end of this first phase, vertices t_0 and t_{J+1} are not yet connected and the rest of the graph is not guaranteed to be fully connected (it could be composed of several disjoint sub-graphs).

- The second phase then fully connects the graph by adding one edge between t_0 and each other task without a predecessor (except t_{J+1}), and one edge between each task without a successor (except t_0) and vertex t_{J+1}.

This produces a connected graph with one single source task (t_0) and one single sink task (t_{J+1}).

Of course, the above specifications work if and only if the number of internal edges n_e is less or equal the theoretical maximum: the J non-artificial tasks have at most two input internal edges except task t_1 which has none and t_2 which has at most one (from t_1). The maximum number of internal edges is thus $0 + 1 + 2 \times (J - 2) = 2 \times J - 3$. Our specifications exclude the generation of graphs with too few internal edges. These graphs would result in problem instances close to the classical multi-dimensional bin packing problem. We considered only graphs with at least one incident internal edge per non-artificial task (i.e., the J tasks included between the source and sink), that is a minimum of $J-1,\ J > 1$. This is the case, for instance, of a perfectly sequential application (or two independent sequential applications with one extra internal edge in one of the two sub-DAG).

With this random generator we produced 37500 different graphs with 6 to 30 tasks, 3 to 53 internal edges, and 5 to 61 total edges. By means of example, Fig. A.23 shows two graphs, without the artificial source and sink tasks, with $J = 10$ tasks, one with $n_e = 7$ internal edges (the minimum) and the other with $n_e = 13$ (the maximum).



(a) 7 internal edges

(b) 13 internal edges

Figure A.23: Randomly generated DAGs with 10 tasks (source and sink are not represented).

Appendix B. MILP formulation

In this appendix we report the complete MILP formulation for our FPGA scheduling problem, that we used in Section 5 to evaluate our heuristic. We remind to the reader that in this appendix, term n_t denotes the total number of tasks in a DAG (nodes). With respect to the notation used in Section 4, $n_t = J + 2$, J being the number of actual tasks (no source and no sink tasks). We start with the input values:

- $n_t \in \mathbb{N},\ n_t \geq 2$ is the total number of tasks, including the artificial source and sink. In the following we denote $\mathfrak{T} = \{T_0, \ldots, T_{n_t-1}\}$ the set of tasks. T_0 and T_{n_t-1} are the artificial source and sink tasks with zero duration and zero resource consumption. We remind that the artificial source and sink tasks are just a way to close the DAG that represents the inter-task dependencies and that they are added without loss of generality. They ease the modeling by providing simple start and end points but they have

no impact on the results.

- $\boldsymbol{y} \in \mathbb{R}^{+^{n_t}}$ is the vector of the durations of the tasks; y_t is the duration of task T_t. We always have $y_0 = y_{n_t-1} = 0$ because tasks T_0 and T_{n_t-1} have zero duration.

- $n_r \in \mathbb{N}, n_r \geq 1$ is the number of types of resources offered by the target FPGA. $\mathfrak{R} = \{R_1, \ldots, R_{n_r}\}$ is the set of types of resources (e.g. LE, DSP blocks, Embedded Memory Blocks, Clock Generators...)

- $n_s = n_t - 2 \in \mathbb{N}$ is the total number of slots. $\mathfrak{S} = \{S_1, \ldots, S_{n_s}\}$ is the set of slots. By definition the maximum number of slots is equal to the number of non-artificial tasks so, to simplify the problem specification, we fix the number of slots to its theoretical maximum ($n_s = n_t - 2$). In most cases the last slots of a schedule have no allocated tasks and are not considered in the computed makespan, but in exceptional cases it can be that all slots are used, each by one single task.

- $\mathcal{T} \in \mathbb{R}^+$ is the reconfiguration time of the target FPGA.

- $P \in \{0,1\}^{n_t \times n_t}$ is a matrix of precedence relations; elements of P are binary values; if P element $\pi_{t,t'} = 1$, then task T_t precedes task $T_{t'}$ ($T_{t'}$ can execute only after T_t termination). Precedence is transitive so all precedence matrices with same transitive closure are equivalent. P is not a free variable and must obey constraints: the directed graph G with vertices in $\mathfrak{T}$ and edges defined by P must be acyclic. P must also be such that task T_0 is a direct or indirect predecessor of all other tasks and task T_{n_t-1} is a direct or indirect successor of all other tasks.

- $\boldsymbol{q} \in \mathbb{N}^{n_r}$ is the vector of the total available quantities of FPGA resources; q_r is the total quantity of resource R_r. We decided to use natural number values because FPGA hardware resources are usually discrete elements. As these are input variables, and no output integer variables are derived from them, this choice has no impact on the solving complexity. If other

types of resources were needed and would be better described by real numbers (e.g. energy), this choice could easily be changed and would have no impact.

- $R \in \mathbb{N}^{n_t \times n_r}$ is the matrix of resources consumptions; elements of R are natural numbers; R element $0 \leq \rho_{t,r} \leq q_r$ is the consumption of resource R_r by task T_t (we consider only problem instances for which solutions exist, so there are no tasks that consume alone more resources than what is available). The artificial source and sink tasks do not consume any resource:

$$\forall\ 1 \leq r \leq n_r\ \rho_{0,r} = \rho_{n_t-1,r} = 0$$

- $\mathcal{H}$ is a large number, larger than every possible makespan. In MILP parlance this is the "horizon". In our case $\mathcal{H}$ can easily be set to the sum of the task durations plus the sum of the maximum number of reconfiguration times. This extreme case corresponds to the worst possible schedule where all slots are used with one single non-artificial task per slot:

$$\mathcal{H} = n_s \times \mathcal{T} + \sum_{0 \leq t=0 < n_t} y_t$$

Outputs

The outputs are the values computed by the MILP solver and that fully define a solution. They must be very carefully selected and their types (real, integer, binary) must also be carefully chosen because these choices have a strong impact on the solving complexity. The theoretical complexity of the general Linear Programming (LP) problem is polynomial and instances can actually be solved in polynomial time using interior-point techniques. Changing some of its real output variables into integer output variables changes the LP problem into MILP and the theoretical complexity becomes NP-hard. The rule of thumb of our formulation is thus to limit the number of integer output variables and use binary variables instead of integer variables whenever possible.

- $\boldsymbol{x} \in \mathbb{R}^{+^{n_t}}$ is the vector of start times of tasks; x_t is the start time of task T_t. Without loss of generality we constrain the start time of the artificial source task to be zero: $x_0 = 0$. The x_0 component is thus technically an input.

- $\boldsymbol{z} \in \mathbb{R}^{+^{n_t}}$ is the vector of end times of tasks; z_t is the end time of task T_t. If we consider the x_t as true output variables, as task durations y_t are known inputs, the z_t are not true output variables: they can be computed from the x_t and the y_t and they should not add to the complexity.

- $\boldsymbol{u} \in \mathbb{R}^{+^{n_s}}$ is the vector of start times of slots; u_s is the start time of slot S_s. Without loss of generality we constrain the start time of the first slot to be zero: $u_1 = 0$. The u_1 component is thus technically an input. This means that the initial reconfiguration time is not counted in the total makespan, which is the same convention used by the heuristics we compare with MILP.

- $\boldsymbol{v} \in \mathbb{R}^{+^{n_s}}$ is the vector of durations of slots; v_s is the duration of slot S_s.

- $\boldsymbol{w} \in \mathbb{R}^{+^{n_s}}$ is the vector of end times of slots; w_s is the end time of slot S_s. A bit like for the x_t, y_t and z_t, the u_s, v_s and w_s are redundant: the w_s, for instance, can be computed from the u_s and the v_s and they should not add to the complexity.

- $A \in \{0,1\}^{n_t \times n_s}$ is the allocation matrix; elements of A are binary values. A element $\alpha_{t,s} = 1$ if task T_t is allocated in slot S_s, else $\alpha_{t,s} = 0$. The $\alpha_{t,s}$ are the unavoidable but only source of MILP solving complexity. Without loss of generality we constrain task T_0 to be allocated to slot S_1: $\alpha_{0,1} = 1$ and $\forall\ 1 < s < n_s, \alpha_{0,s} = 0$. These matrix elements are thus technically inputs.

Objective function

Our objective function is the total makespan and can be very easily expressed using the output variables: it is $z_{n_t-1} - x_0 = z_{n_t-1}$, the end time of the

artificial sink task T_{n_t-1}. The MILP solver will be instructed to find a solution that complies with the constraints and that minimizes z_{n_t-1}. This objective function does not involve the number of actually used slots, which naturally solves the potential issue about the total number of slots $n_s = n_t - 2$; the unused slots, if any, and the corresponding reconfiguration times are not counted in the objective function and do not influence the optimization effort.

Constraints

We present here a high-level human-readable form of the constraints of the MILP formulation. The low-level form that can directly be used by the solver is less intuitive. It makes use of auxiliary variables defined by linear equations of the input and output variables. A low-level constraint then consists in specifying upper and/or lower bounds of the output and auxiliary variable, plus type constraints on the output variables (e.g. the $\alpha_{t,s}$ are binary values).

For instance, in order to express that a task T_t cannot be pre-empted and its end time is its start time plus its duration we introduce the auxiliary variables a_t such that:

$$\forall\ 0 \leq t < n_t,\ a_t = x_t + y_t - z_t$$
$$\forall\ 0 \leq t < n_t,\ 0 \leq a_t \leq 0$$

The list of time-related constraints is the following:

- A task cannot be pre-empted, its end time is equal to its start time plus its duration:

$$\forall\ 0 \leq t < n_t,\ z_t = x_t + y_t$$

- There is no idle time between slots and the start time of a slot is the end time of the previous slot plus the reconfiguration time:

$$\forall\ 2 \leq s \leq n_s,\ u_s = w_{s-1} + \mathcal{T}$$

- A slot cannot be pre-empted, its end time is equal to its start time plus its duration:

$$\forall\ 1 \leq s \leq n_s,\ w_s = u_s + v_s$$

- If a task T_t is allocated in a slot S_s ($\alpha_{t,s} = 1$), then its execution happens during the slot's duration:

$$\forall\ 0 \leq t < n_t,\ \forall\ 1 \leq s \leq n_s,\ \alpha_{t,s} = 1 \Rightarrow u_s \leq x_t \leq z_t \leq w_s$$

- If a task T_t precedes another task T'_t ($\pi_{t,t'} = 1$), then T_t end time is less or equal $T_{t'}$ start time:

$$\forall\ 0 \leq t < n_t - 1,\ \forall\ 1 \leq t' < n_t,\ \pi_{t,t'} = 1 \Rightarrow z_t \leq x_{t'}$$

The resources-related constraints can be condensed in one single statement: the tasks allocated to a slot cannot use more of any resource than its total available quantity:

$$\forall\ 1 \leq r \leq n_r,\ \forall\ 1 \leq s \leq n_s,\ \sum_{0 \leq t < n_t, \alpha_{t,s}=1} \rho_{t,r} \leq q_r$$

Finally one more constraint is needed to express the fact that a task is allocated to one and only one slot:

$$\forall\ 0 \leq t < n_t,\ \exists!\ 1 \leq s \leq n_s, \alpha_{t,s} = 1$$

Appendix C. Correcting the pseudo-code of Highest Priority First, Next-Fit (HPF-NF)

Algorithm 6 shows the pseudo-code of heuristic HPFNF as published in [4]. The aim of HPFNF is to group tasks (nodes in a dependency graph $G = \langle N, E \rangle$) in reconfiguration stages by considering tasks sorted in a topological priority where parent tasks have higher priority over child tasks (Algorithm 5). HPF-NF assigns a task to a reconfiguration stage S_k only if it fits resources available in S_k and if all of its parent tasks have been scheduled into the same stage or in previous stages (line 12 in Algorithm 6). Otherwise, the algorithm attempts to schedule tasks with lower priorities or creates a new reconfigurations stage.

Note that, differently from our work, the pseudo-code of HPF-NF, considers a dependency graph G without source and sink nodes. Unfortunately, the pseudo-code in Algorithm 6 does not behave as described above for the following two reasons:

- At line 12, in Algorithm 6, the second part of the if statement checks the visited status of a task's parents (immediate predecessors). This condition does not express the requirement that a task be scheduled in stage S_k only if its predecessors have been scheduled in stages $S_j,\ j = k, k-1, k-2, ..., 1$. Instead, this part of the if statement states that a necessary condition for a task to be added to S_k is that its parents have already been *considered* for addition. However, parent tasks may have already been considered for addition, but discarded (thus remaining unscheduled) because their resource occupancy didn't fit the available resources of S_k.

- At line 16, in Algorithm 6, the body of the else statement implies that a task is marked as visited only if it cannot be inserted in a reconfiguration stage S_k. This prevents the first part of the if statement at line 6 to be ever evaluated to true. As a consequence, a new stage can only be instantiated if, by chance, the resource occupancy of all tasks in the current stage S_k match exactly the resource availability W (second part of the if statement at line 6).

The corrected pseudo-code is shown in Algorithm 7, where the corrected portions of pseudo-code are highlighted in yellow:

- A task is added to a reconfiguration stage S_k if it fits the available resources of S_k *and* all of its parents have already been scheduled, hence they have been removed from the worklist O.
- A task is always marked as visited, after it has been considered for inclusion in a reconfiguration stage.

In Fig. C.24 and Fig. C.25, we respectively show the same example dependency graph and the resulting scheduling as those published in [4]. Here, tasks consume a single (normalized) resource (annotated in the graph) and the target FPGA is assumed to dispose of 100 elements of such a normalized resource ($W = 100$). The priority of tasks is annotated in Fig. C.25, where shadowed shapes denote the reconfiguration stages.

```
1 Function sort( dependency graph G = < N, E > ):
2     Initialize set O with N;
3     Initialize queue Q to empty;
4     while O is not empty do
5         Find the lightest node in the current top level of G and assume it is node x;
6         Put node x into queue Q, remove x from O and from G;
7     end
8     return Q;
```

Algorithm 5: The pseudo-code of the algorithm to assign a topological priority to tasks in a dependency graph, [4]. A set of tasks (nodes) in G have the same level l_i if they have the same maximum distance from G's source. Lightness of a node is defined with respect to its resource occupancy: the lightest node among a set of candidates is the one that occupies the least (normalized) resources.

```
1  Function hpfnf( dependency graph G = < N, E > ):
2      Store in worklist O the result of sort(G);/* See Algorithm 5        */
3      Set the status of nodes in O unvisited;
4      k = 1;/* Stage index                                                 */
5      while O is not empty do
6          if All nodes are visited or sum_{i in S_k} w_i = W then
7              k = k + 1;/* Create a new reconfiguration stage             */
8              Set the status of nodes in O unvisited;
9          end
10         else
11             Find the highest priority node from unvisited nodes and assume it is x;
12             if sum_{i in S_k} w_i + w_x <= W and node x has no visited parent nodes
                then
13                 Put node x into S_k and remove it from O and G;
14             end
15             else
16                 Set the status of node x visited;
17             end
18         end
19     end
```

Algorithm 6: The pseudo-code for heuristic High Priority First, Next-Fit (HPF-NF) as published in [4]

```
1  Function hpfnfCorrect( dependency graph G = < N, E > ):
2      Store in worklist O the result of sort(G);/* See Algorithm 5        */
3      Set the status of nodes in O unvisited;
4      k = 1;/* Stage index                                                 */
5      while O is not empty do
6          if All nodes are visited or ∑_{i∈S_k} w_i = W then
7              k = k + 1;/* Create a new reconfiguration stage             */
8              Set the status of nodes in O unvisited;
9          end
10         else
11             Find the highest priority node from unvisited nodes and assume it is x;
12             if ∑_{i∈S_k} w_i + w_x ≤ W and node x has no parent nodes in O then
13                 Put node x into S_k and remove it from O and G;
14             end
15             Set the status of node x visited;
16         end
17     end
```

Algorithm 7: The correct pseudo-code for heuristic HPF-NF, Algorithm 6; corrections are highlighted in yellow.

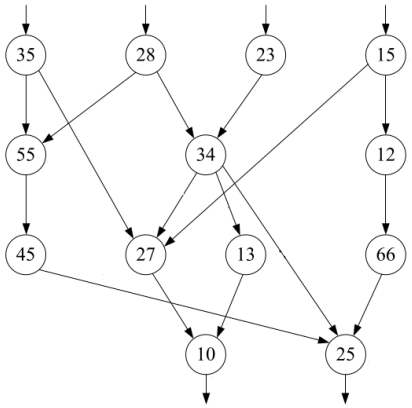


Figure C.24: The example dependency graph G from [4] (source and sink are not part of G in [4]). The numbers within nodes denote the normalized resource consumption. Based on the dependencies, four levels are present: $l_1 = \{35, 28, 23, 15\}$, $l_2 = \{55, 34, 12\}$, $l_3 = \{45, 27, 13, 66\}$, $l_4 = \{10, 25\}$.

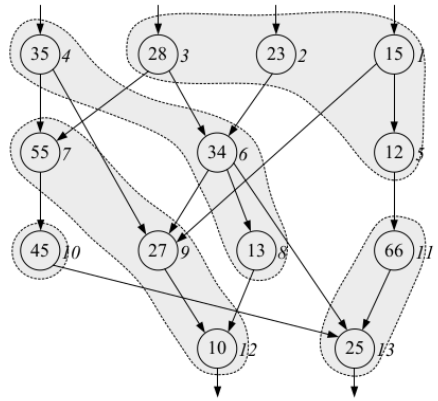


Figure C.25: The dependency graph in Fig. C.24 scheduled by Algorithm 7 for a target FPGA with $W = 100$, as published in [4]. Task nodes are annotated with the priority assigned by Algorithm 5, 1 is the highest priority, 13 the lowest. Reconfiguration stages are $S_1 = \{28, 23, 15, 12\}$, $S_2 = \{35, 34, 13\}$, $S_3 = \{55, 27, 10\}$, $S_4 = \{45\}$, $S_5 = \{66, 25\}$.

References

[1] P. Mell, T. Grace, The NIST Definition of Cloud Computing, `http://csrc.nist.gov/publications/detail/sp/800-145/final`, last visited on December 2020.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia, A View of Cloud Computing, Commun. ACM 53 (4) (2010) 50–58.

[3] S. Asano, T. Maruyama, Y. Yamaguchi, Performance comparison of fpga, gpu and cpu in image processing, in: FPL, 2009, pp. 126–131.

[4] M. Huang, H. Simmler, P. Saha, T. El-Ghazawi, Hardware task scheduling optimizations for reconfigurable computing, in: HPRCTA, 2008, pp. 1–10.

[5] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, K. Wang, Enabling FPGAs in the Cloud, in: CF, 2014.

[6] S. Zeitouni, G. Dessouky, A. Sadeghi, SoK: On the Security Challenges and Risks of Multi-Tenant FPGAs in the Cloud, CoRR abs/2009.13914.

[7] I. Kuon, R. Tessier, J. Rose, FPGA Architecture: Survey and Challenges, Foundations and Trends in Electronic Design Automation 2 (2007) 135–253.

[8] T. D. A. Nguyen, A. Kumar, Maximizing the Serviceability of Partially Reconfigurable FPGA Systems in Multi-Tenant Environment, in: FPGA, 2020, pp. 29–39.

[9] T. Xia, j.-c. Prvotet, F. Nouvel, Hypervisor mechanisms to manage FPGA reconfigurable accelerators, in: FTP, 2016, pp. 44–52.

[10] M. Najem, T. Bollengier, J.-C. L. Lann, L. Lagadec, Extended overlay architectures for heterogeneous FPGA cluster management, Journal of Systems Architecture 78 (2017) 1–14.

[11] A. Brant, G. G. F. Lemieux, ZUMA: An Open FPGA Overlay Architecture, in: FCCM, 2012, pp. 93–96.

[12] D. Koch, C. Beckhoff, G. G. F. Lemieux, An efficient FPGA overlay for portable custom instruction set extensions, in: FPL, 2013, pp. 1–8.

[13] G. Stitt, J. Coole, Intermediate Fabrics: Virtual Architectures for Near-Instant FPGA Compilation, IEEE Embedded Systems Letters (2011) 81–84.

[14] A. K. Jain, D. L. Maskell, S. A. Fahmy, Are Coarse-Grained Overlays Ready for General Purpose Application Acceleration on FPGAs?, in: DASC, 2016, pp. 586–593.

[15] K. Rosvall, I. Sander, A constraint-based design space exploration framework for real-time applications on mpsocs, in: DATE, 2014, pp. 1–6.

[16] Intel, CTAccel Image Processing (CIP) accelerator for Intel PAC, `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/solution-sheets/ctaccell-solution-brief.pdf`.

[17] E. A. Lee, D. G. Messerschmitt, Synchronous data flow, Proceedings of the IEEE 75 (9) (1987) 1235 – 1245.

[18] F. Habibi, F. Barzinpour, S. Sadjadi, Resource-constrained project scheduling problem: review of past and recent developments, Journal of Project Management 3 (2) (2018) 55–88.

[19] J. Blazewicz, J. Lenstra, A. Kan, Scheduling subject to resource constraints: classification and complexity, Discrete Applied Mathematics 5 (1) (1983) 11–24.

[20] Y. Qu, J.-P. Soininen, J. Nurmi, Static Scheduling Techniques for Dependent Tasks on Dynamically Reconfigurable Devices, J. Syst. Archit. 53 (11) (2007) 861–876.

[21] E. A. Deiana, M. Rabozzi, R. Cattaneo, M. D. Santambrogio, A multi-objective reconfiguration-aware scheduler for FPGA-based heterogeneous architectures, in: ReConFig, 2015, pp. 1–6.

[22] C. Y. Lin, N. Wong, H. K.-H. So, Operation Scheduling and Architecture Co-Synthesis for Energy-Efficient Dataflow Computations on FPGAs, in: FPGA, 2012, p. 270.

[23] W. Fu, K. Compton, Scheduling intervals for reconfigurable computing, in: FCCM, 2008, pp. 87–96.

[24] Y. Zhao, C. Tian, Z. Zhu, J. Cheng, C. Qiao, A. X. Liu, Minimize the Makespan of Batched Requests for FPGA Pooling in Cloud Computing, IEEE Transactions on Parallel and Distributed Systems 29 (11) (2018) 2514–2527.

[25] D. Merkle, M. Middendorf, H. Schmeck, Ant colony optimization for resource-constrained project scheduling, IEEE Transactions on Evolutionary Computation 6 (4) (2002) 333–346.

[26] C. Jing, Y. Zhu, M. Li, Energy-Efficient Scheduling on Multi-FPGA Reconfigurable Systems, MICRO 37 (6-7) (2013) 590–600.

[27] F. Abdallah, C. Tanougast, I. Kacem, C. Diou, D. Singer, Genetic algorithms for scheduling in a CPU/FPGA architecture with heterogeneous communication delays, Comput. Ind. Eng. 137.

[28] T. L. Adam, K. M. Chandy, J. R. Dickson, A comparison of list schedules for parallel processing systems, Commun. ACM 17 (12) (1974) 685690.

[29] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. on Par. and Dist. Sys. 13 (3) (2002) 260–274.

[30] A. Guillaume, S. Manu, B. Anne, R. Yves, R. Padma, Co-scheduling algorithms for high-throughput workload execution, Journal of Scheduling 19 (6) (2016) 627–640.

[31] K. Danne, M. Platzner, Partitioned scheduling of periodic real-time tasks onto reconfigurable hardware, in: IPDPS, 2006, p. 8.

[32] J. chiou Liou, M. A. Palis, An efficient task clustering heuristic for scheduling dags on multiprocessors, in: Workshop on Resource Management at IPDPS, 1996, pp. 152–156.

[33] G. C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 3rd Edition, Springer, 2011.

[34] K. Danne, R. Mühlenbernd, M. Platzner, Server-based execution of periodic tasks on dynamically reconfigurable hardware, IET Computers & Digital Techniques 1 (2007) 295–302.

[35] M. Götz, Run-Time Reconfigurable RTOS for Reconfigurable Systems-on-Chips, Ph.D. thesis, Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn (2007).

[36] M. Götz, F. Dittmann, C. Pereira, Deterministic Mechanism for Run-time Reconfiguration Activities in an RTOS, in: INDIN, 2006, pp. 693–698.

[37] A. Lodi, S. Martello, D. Vigo, Recent Advances on Two-Dimensional Bin Packing Problems, Discrete Appl. Math. 123 (1-3) (2002) 379–396.

[38] Z. Gu, W. Liu, J. Xu, J. Cui, X. He, Q. Deng, Efficient Algorithms for 2D Area Management and Online Task Placement on Runtime Reconfigurable FPGAs, MICRO 33 (5-6) (2009) 374–387.

[39] H. M. Salkin, C. A. D. Kluyver, The knapsack problem: A survey, Naval Research Logistics Quarterly 22 (1) (1975) 127–144.

[40] M. Bertolino, A. Enrici, R. Pacalet, L. Apvrille, Efficient scheduling of fpgas for cloud data center infrastructures, in: DSD, 2020, pp. 57–64.

[41] Xilinx, Ultrascale+ FPGAs Product Tables and Product Selection Guide, `https://www.xilinx.`

com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf.

[42] J. Gross, J. Yellen, M. Anderson, Graph Theory and its Applications, 3rd Edition, CRC Press, 2018.

[43] K. Paul, C. Dash, M. S. Moghaddam, reMORPH: A Runtime Reconfigurable Architecture, in: DSD, 2012, pp. 26–33.

[44] G. J. Brebner, A Virtual Hardware Operating System for the Xilinx XC6200, in: FPL, 1996, p. 327336.

[45] G. Charitopoulos, I. Koidis, K. Papadimitriou, D. Pnevmatikatos, Hardware task scheduling for partially reconfigurable fpgas, in: ARC, 2015, pp. 487–498.

[46] K. Dang Pham, E. Horta, D. Koch, BITMAN: A tool and API for FPGA bitstream manipulations, in: DATE, 2017, pp. 894–897.

[47] M. L. Silva, J. C. Ferreira, Creation of Partial FPGA Configurations at Run-Time, in: DSD, 2010, pp. 80–87.

[48] A. Laleve, P. Horrein, M. Arzel, M. Hbner, S. Vaton, AutoReloc: Automated Design Flow for Bitstream Relocation on Xilinx FPGAs, in: DSD, 2016, pp. 14–21.

[49] GLPK, https://www.gnu.org/software/glpk/ (2012).

[50] G. Csardi, T. Nepusz, S. Horvat, V. Traag, F. Zanini, https://igraph.org/ (2020).

[51] Yang Qu, J. . Soininen, J. Nurmi, A Parallel Configuration Model for Reducing the Run-Time Reconfiguration Overhead, in: DATE, 2006, pp. 1–6.

[52] J. Resano, D. Verkest, D. Mozos, S. Vernalde, F. Catthoor, A hybrid design-time/run-time scheduling flow to minimise the reconfiguration overhead of FPGAs, MICRO 28 (5) (2004) 291–301.